Lecture Notes in Artificial Intelligence 1871
Subseries of Lecture Notes in Computer Science

James L. Rash Christopher A. Rouff
Walt Truszkowski Diana Gordon
Michael G. Hinchey (Eds.)

Formal Approaches to Agent-Based Systems

First International Workshop, FAABS 2000
Greenbelt, MD, USA, April 2000
Revised Papers

Springer

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

James L. Rash   Christopher A. Rouff
Walt Truszkowski   Diana Gordon
Michael G. Hinchey (Eds.)

Formal Approaches to Agent-Based Systems

First International Workshop, FAABS 2000
Greenbelt, MD, USA, April 5-7, 2000
Revised Papers

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

James L. Rash
Walt Truszkowski
Michael G. Hinchey
NASA Goddard Space Flight Center
Greenbelt, Maryland 20771, USA
E-mail: {james.rash/walt.truszkowski/mike.hinchey}@gsfc.nasa.gov

Christopher A. Rouff
Science Applications International Corporation
4001 Fairfax Drive, Suite 300, Airlington, VA 22203, USA
E-mail: rouffc@saic.com

Diana Gordon
Naval Research Laboratory
4555 Overlook Avenue, S.W., Washington, D.C., USA
E-mail: gordon@aic.nrl.navy.mil or dspears@cs.uwyo.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Formal approaches to agent based systems : first international workshop ; revised papers / FAABS 2000, Greenbelt, MD, USA, April 5 - 7, 2000. James L. Rash ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 1871 : Lecture notes in artificial intelligence)
ISBN 3-540-42716-3

CR Subject Classification (1998): I.2.11, I.2, C.2.4, F.3, F.4.1, D.2.4

ISBN 3-540-42716-3 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10722329 06/3142 5 4 3 2 1 0

Preface

The FAABS Workshop was conceived at NASA Goddard Space Flight Center while the Agent Technology Development Group in the Advanced Architectures and Automation Branch (Code 588) was developing a prototype agent community to automate satellite ground operations. During the development of this system, several race conditions arose within and between agents. Due to the complexity of the agents and the communications between them, it was decided that a formal approach was needed to specify the agents and the communications between them, so that the system could be checked for additional errors.

A formal model of the inter-agent communications was developed, with the expectation that this would enable us to find more errors. Success on this convinced us of the importance of using formal methods to model agent-based systems. To share our own experiences and to learn how others were approaching these issues, we decided to hold a workshop on formal methods and agent-based systems.

The response was overwhelming. We intentionally limited numbers to encourage discussion and interaction. Invitations were issued on the basis of either a paper abstract, or a position statement explaining the author's interests and experience. The response was so great that we had to increase the number of invitations dramatically, and limit participation to one author per paper.

The workshop was a successful gathering of people from all over the world to discuss formal approaches to agent-based systems. The wide range of approaches and many interesting interactions of the participants made the workshop a rewarding experience. Posters, paper presentations, panels, and the invited talk by J Moore stimulated much discussion.

The proceedings contains both papers and write-ups of the poster presentations and panel discussions. It also includes author contact information to help further information sharing.

We would like to express our sincere thanks to all those who attended the workshop, presented papers or posters, and participated in panel sessions and both formal and informal discussions. Our thanks go to the NASA Goddard Code 588 for financing the workshop. Additionally, thanks go to the Naval Research Laboratory and to DARPA for their active support of the workshop.

Our thanks also go to Springer-Verlag for their assistance and interest in publishing the proceedings of this first Workshop on Formal Approaches to Agent-Based Systems.

July 2001

James L. Rash
Christopher A. Rouff
Walter Truszkowski
Diana F. Gordon

Table of Contents

Model Checking-Based Analysis of Multiagent Systems

Massimo Benerecetti[1] and Fausto Giunchiglia[2]

[1] Dept. of Phisical Sciences, Università di Napoli "Federico II", Italy
[2] DISA - Università degli Studi di Trento, Italy

Abstract. Model checking is a very successful technique which has been applied in the design and verification of finite state concurrent reactive processes. This technique can be lifted to be applicable to multiagent systems. Our approach allows us to reuse the technology and tools developed in model checking, to design and verify multiagent systems in a modular and incremental way, and also to have a very efficient model checking algorithm. The paper investigates the applicability of our multiagent model checking framework to the analysis of a relevant class of multiagent protocols, namely security protocols.

1 Introduction

Model checking is a very successful automatic technique which has been devised for the design and verification of finite state reactive systems, e.g., sequential circuit designs, communication protocols, and safety critical control systems (see, e.g., [5]). There is evidence that model checking, when applicable, is far more successful than the other approaches to formal methods and verification (e.g., first order or inductive theorem proving, tableau based reasoning about modal satisfiability). Nowadays many very efficient and well developed tools implementing model checking, so-called "model checkers", are available (for instance: SPIN [7] and, NuSMV [4]). Some these tools have been successfully applied to real case studies, and are currently in use in industry.

As shown in [2], model checking can be "lifted" to multiagent systems in a way to *(i)* reuse with almost no variations all the technology and tools developed in model checking; and *(ii)* allow for a modular design and verification of multiagent systems. The first feature allows us to exploit the huge amount of expertise, technology and tools developed in model checking. The second feature is necessary in order to deal with real world complex systems.

In the paper we investigate the applicability of the model checking framework to the analysis of a relevant class of multiagent protocols, namely security protocols. Security protocols can be seen as particular multiagent protocols in which agents exchange messages and need to ensure some security properties of their communication. In particular we show how the framework can be applied to automate analysis of protocols by means of an example. The example chosen is a well known security protocol, the Andrew protocol, which, in its original version, is known to be faulty (see [3]).

The paper is structured as follows: Sections 2 and 3 briefly present the multiagent specification framework and the multiagent model checking algorithm. Section 4 starts by presenting the Andrew protocol first; then it illustrates a belief-based specification of the protocol in our framework; following we describe how the model checking algorithm works, showing that it is able to discover the bug in the protocol. Finally, we analyze the corrected version of the protocol (as proposed in [3]), showing that the algorithm correctly proves that this version doesn't exhibit the bug.

2 The Multiagent Specification Framework

We build the notion of agent incrementally over the notion of process. Suppose we have a set I of agents and $O = \{\mathbf{B}, \mathbf{D}, \mathbf{I}\}$ is a set of symbols, one for each BDI attitude. Each agent is seen as a process having its own beliefs, desires and intentions about (itself and) other agents. We adopt the usual syntax for beliefs: $B_i\phi$ ($D_i\phi$, $I_i\phi$, respectively) means that agent i believes (desires, intends, respectively) ϕ, and ϕ is a belief (desire, intention, respectively) of i. We will denote with O_i any one of the BDI operators B_i, D_i, I_i (for i).

The idea is to associate to each level of nesting of BDI operators a process evolving over time. Therefore, let $OI = \{\mathbf{O}1, ..., \mathbf{O}n\}$, where each index $1, ..., n \in I$ corresponds to an agent in the system, while $\mathbf{O}$ denotes any one of $\mathbf{B}, \mathbf{D}, \mathbf{I}$. The set OI^* will denote the set of finite (possibly empty) strings of the form $\mathbf{O}1, ..., \mathbf{O}n$ with $\mathbf{O}i \in OI$. We call any $\alpha \in OI^*$, a *view*. Each view in OI^* corresponds to a possible nesting of BDI operators. We also allow for the empty string, ϵ.

Figure 1 graphically represents this situation. The intuition is that ϵ represents the view of an external observer (e.g., the designer) which, from the outside, "sees" the behavior of the overall system. To each nesting of BDI operators we associate a view of the corresponding agent. Intuitively, in Figure 1, the beliefs of agent 1 correspond to the view $\mathbf{B}1$ and can be modeled by a process playing 1's role in the system. The beliefs that 1 has about the intentions of agent 2 correspond to the view $\mathbf{B}1\mathbf{I}2$ and can be modeled by a process modeling 2's intentions in (1's view of) the system. Things work in the same way for the BDI attitudes of 2, those that 2 ascribes to 1, and so on. An agent is thus a tree of views rooted in the view that the external observer has of it, e.g., $\mathbf{B}1$.

We associate a logical language $\mathcal{L}_\alpha$ to each view $\alpha \in OI^*$. Intuitively, each $\mathcal{L}_\alpha$ is the language used to express what is true (and false) in the process of view α. In order to express properties of processes in each view we employ the logic CTL. For each α, P_α is a set of propositional atoms. Each P_α allows for the definition of a different language, called a MultiAgent Temporal Logic (MATL) language (on P_α). A MATL language $\mathcal{L}_\alpha$ on P_α is the smallest CTL language containing the set of propositional atoms P_α and the BDI atom $\mathrm{O}_i\phi$ for each formula ϕ of $\mathcal{L}_{\alpha\mathbf{O}i}$. In particular, $\mathcal{L}_\epsilon$ is used to speak about the whole system. The language $\mathcal{L}_{\mathbf{B}i}$ is the language adopted to represent i's beliefs. The language $\mathcal{L}_{\mathbf{B}i\mathbf{I}j}$ is used to specify i's beliefs about j's intentions, and so on. Intuitively, the formula $\mathsf{AG}\,(p \supset B_i\neg q) \in \mathcal{L}_\epsilon$, (denoted by $\epsilon : \mathsf{AG}\,(p \supset B_i\neg q)$) means that in

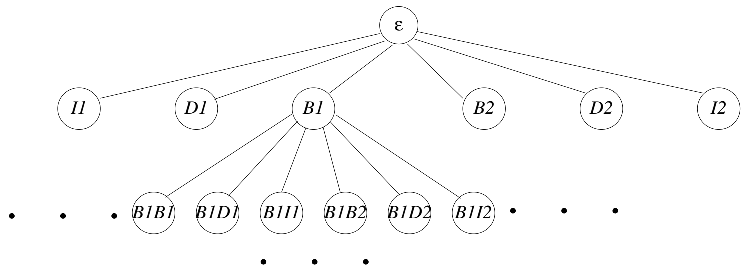


Fig. 1. The tree of views.

every future state of the external view ϵ, if p is true then agent i believes that q is false.

We are interested in extending CTL model checking to the model checking of BDI formulae. In model checking, finite state processes are modeled as finite state machines. A *finite state machine* (FSM) is a tuple $f = \langle S, J, R, L\rangle$, where S is a finite set states, $J \subseteq S$ is the set of *initial states*, R is a total binary relation on S, and $L : S \to \mathcal{P}(P)$ is a *labeling function*, which associates to each state $s \in S$ the set $L(s)$ of propositional atoms true at s. Our solution is to extend the notion of FSM to that of *MultiAgent Finite State Machine (MAFSM)*, where, roughly speaking, a MAFSM is a finite set of FSMs.

A first step in this direction is to restrict ourselves to a finite number of views α. Thus, let OI^n denote a finite subset of OI^* obtained by taking the views in any finite subtree of OI^* rooted at view ϵ. This restriction is not enough, as a finite set of views still allows for an infinite number of BDI atoms. Even if we had a finite number of processes we would not be able to model them as FSMs. This problem can be solved introducing the notion of *explicit BDI atom* (in symbols $Expl(\mathbf{O}i, \alpha)$) as a finite subset of BDI atoms. Those are the only BDI atoms which are explicitly represented (together with the propositional atoms) in a FSM.[1]

In order to give a notion of satisfiability in a MAFSM, we start from the notion of satisfiability of CTL formulae in an FSM at a state (defined as in CTL structures).

Since the FSMs in each view are built on the propositional and explicit BDI atoms of the view, to assess satisfiability of all the propositional and explicit BDI atoms (and all the CTL formulae build out of them) we do not need to use the machinery associated to BDI operators. Their truth value can be computed using the standard notion of CTL satisfiability. However, this machinery is needed in order to deal with the (infinite) number of BDI atoms which are not memorized anywhere in MAFSM.

[1] In general, there may be more than one FSM associate to each view. This allows for situations in which a view can be only partially specified, and consequently there can be more than one process modeling that view.

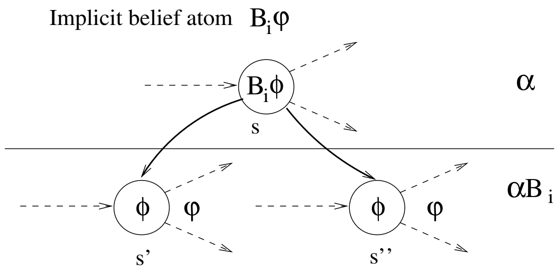


Fig. 2. Explicit BDI atoms and satisfiability.

Let the set of *implicit BDI atoms* of a view α, written $Impl(\mathbf{O}i, \alpha)$, be defined as the (infinite) subset of all BDI atoms of $\mathcal{L}_\alpha$ which are not explicit BDI atoms, i.e., $Impl(\mathbf{O}i, \alpha) = \{\mathrm{O}_i\phi \in \mathcal{L}_\alpha \setminus Expl(\mathbf{O}i, \alpha)\}$. The idea is to use the information explicitly contained in the labeling function of each state s of a FSM f of a view α to assess the truth value of the implicit BDI atoms at a state s. Figure 2 illustrates the inderlying intuition, considering the case of beliefs of agent i only. Intuitively, the agent modeled by FSM f (in view α), when in state s, ascribes to agent i all the explicit BDI atoms of the form $B_i\phi$ true at s. This means that the FMSs of view $\alpha\mathbf{B}i$, which represents the beliefs of i, must be in any of the states (s' and s'') in which the formulae ϕ, occurring as arguments of the explicit BDI atoms, are all true. This motivates the following definitions. Let $ArgExpl(\mathbf{O}i, \alpha, s)$ be defined as follows:

$$ArgExpl(\mathbf{O}i, \alpha, s) = \{\phi \in \mathcal{L}_{\alpha\mathbf{O}i} \mid \mathrm{O}_i\phi \in L(s) \ \cap \ Expl(\mathbf{O}i, \alpha)\}$$

$ArgExpl(\mathbf{O}i, \alpha, s)$ consists of all the formulae $\phi \in \mathcal{L}_{\alpha\mathbf{O}i}$ such that the explicit BDI atom $\mathrm{O}_i\phi$ is true at state s (i.e., belongs to the labeling function of s). They are the formulae which identify the states in which the FSMs modeling view $\alpha\mathbf{O}i$ can be, whenever the process in view α is in state s.

Given the notions defined above, we can define satisfiability of implicit BDI atoms. Let $\mathrm{O}_i\psi$ be an implicit BDI atom of a view α. For each state s of a FSM of α, we can compute $ArgExpl(\mathbf{O}i, \alpha, s)$. As shown in Figure 2, we then check whether FSMs of view $\alpha\mathbf{B}i$, which satisfy $ArgExpl(\mathbf{B}i, \alpha, s)$, also satisfy the argument ψ of the implicit BDI atom $B_i\psi$. If this is the case, then s satisfies $B_i\psi$.[2]

3 The Model-Checking Algorithm

The basic operation of a standard CTL model checking algorithm is to extend the labeling function of an FSM (which considers only propositional atoms) to all the (atomic and not atomic) subformulae of the formula being model checked. In order to perform model checking on a FSM of a MAFSM we need to compute

[2] Notice that this gives to the BDI operators the same strength as modal $K(3m)$, where m is the number of agents.

the truth value of all the implicit BDI atoms occurring in the formula to be checked. Their truth value is not explicitly defined in a FSM of a view, but the notion of satisfiability in a MAFSM tells us how to fix the problem.

The crucial observation is that $ArgExpl(\mathbf{o}_i, \alpha, s)$ is generated from the formulae in $Expl(\mathbf{o}_i, \alpha)$ and the labeling functions of the FSMs in α; $ArgExpl(\mathbf{o}_i, \alpha, s)$ is a finite set; and it only depends on the MAFSM specified (and thus independent from the formula to be model checked).

The model checking algorithm MAMC-View(α, Γ) (see [2] for a complete description and a proof of correctness) takes two arguments: a view α, and a set of MATL formulae $\Gamma \subset \mathcal{L}_\alpha$. MAMC-View$(\alpha, \Gamma)$ performs the following phases:
Phase A. MAMC-View(α, Γ) considers in turn the BDI operators O_i. For each of them, the set $ArgImpl(\mathbf{o}_i, \alpha, \Gamma)$ of all the formulae ϕ which are arguments of the implicit BDI atoms $\mathrm{O}_i\phi$ occurring in Γ, is computed.
Phase B. MAMC-View(α, Γ) calls itself recursively on the view below (e.g., $\alpha\mathbf{o}_i$) and on the sets $ArgImpl(\mathbf{o}_i, \alpha, \Gamma)$. In this process, the algorithm recursively descends the tree structure which needs to be model checked. The leaves of this tree are the views for which there is no need to model check implicit BDI atoms, as there are no more implicit BDI atoms occuring in Γ.
Phase C. This phase is a loop over all the FSMs f of the current view α to extend the labeling functions of the visited FSMs. This loop iteratively performs the following two phases:
Phase C.1. In this phase, all the states of f of the current view α, where the algorithm is, are labeled with the implicit BDI atoms. This phase is executed only if there occur implicit BDI atoms in the input formulae. The labeling of the states of f is computed according to the definition of satisfiability of implicit BDI atoms in a MAFSM. Therefore, for each reachable state s of f, the set $L(s) \cap Expl(\mathbf{o}_i, \alpha)$ is computed. Then, MAMC-View(α, Γ) must compute the implicit BDI atoms occurring in Γ that can be added to the labeling function of s, according to the notion of satisfiability given above. Therefore, for each implicit BDI atom $\mathrm{O}_i\phi$ in Γ, $\mathrm{O}_i\phi$ is added to $L(s)$ only if ϕ is satisfied by all the pairs $\langle f', s' \rangle$ that satisfy the arguments of all the explicit BDI atoms true at s, namely all the pairs that satisfy the formulae in $ArgExpl(\mathbf{o}_i, \alpha, s)$.
Phase C.2. This phase simply calls a standard CTL model algorithm on the FSM f of the current view. Indeed, at this point every state s in the current FSM f is labeled (by phase C.1) with all the atoms (i.e, propositional atoms, explicit and implicit BDI atoms) occurring in the input formulae. Notice that in phase C.2 we can employ any model checker as a black box.

4 A Belief-Based Analysis of an Authentication Protocol

As an example of a multiagent scenario, we consider in this section a simple authentication protocol, known as the Andrew protocol, in which two agents interact with each other to safely exchange a secret encryption key. The protocol has been proved to be vulnerable to various attacks (see, e.g., [3]). In the rest of the paper, we will show how this simple protocol can be modeled and analyzed in our framework both in its original (buggy) version and in the corrected one.

The protocol involves two agents, A and B, which share a secret key K_{ab}, and carry out a handshake to authenticate each other. The ultimate goal of the protocol is to exchange a new secret session key K'_{ab} between A and B. B is intended to be the key server, while A is the recipient.

We use standard notation, and present the version of the protocol proposed in [3]. N_i denotes a nonce (a fresh message) newly created by agent i for the current session; K_{ij} is a shared key between agents i and j; $\{M\}_{K_{ij}}$ denotes a message M encrypted with the key K_{ij}; M_1, M_2 is the message resulting from the concatenation of the two messages M_1 and M_2; while $i \rightarrow j : M$ denotes the fact that principal i sends the message M to agent j. The Andrew protocol can be formulated as follows:

$$\begin{array}{lll} 1 \;\; A \rightarrow B & : & \{N_a\}_{K_{ab}} \\ 2 \;\; B \rightarrow A & : & \{N_a, N_b\}_{K_{ab}} \\ 3 \;\; A \rightarrow B & : & \{N_b\}_{K_{ab}} \\ 4 \;\; B \rightarrow A & : & \{K'_{ab}, N'_b\}_{K_{ab}} \end{array}$$

Intuitively, the protocol works as follows: with message 1, A sends B the (fresh) nonce N_a encrypted with the key K_{ab}, which is supposed to be a good key. The goal of this message is to request authentication from B. With message 2, B sends back to A the nonce N_a concatenated with a newly created nonce N_b, both encrypted. At this point, since B must have decrypted message 1 to be able to generate message 2, A knows that it is talking with B. Then, in message 3, A sends back to B N_b encrypted. This allows B to conclude that it is actually talking to A (as it must have decrypted message 2 to obtain N_b and generate message 3). The two agents are now authenticated. Finally with message 4, B sends A the new session key K'_{ab} together with a new nonce N'_b encrypted with the shared key. The final message is the one subject to attacks. Indeed, in message 4 there is nothing that A can recognize as fresh. An intruder might send A an old message, possibly containing a compromised key.

The kind of analysis we're interested in is based on the idea, originally proposed in [3] (but see also [1]), of studying how messages sent and received during a protocol session by a trusted party may affect its beliefs about the other parties. In the present case, one might want to prove the following property: after message 4, A (respectively B) believes that B (respectively A) believes that the new key is a good key for communication between A and B. This property ensures that both agents believe that the protocol ended correctly and that they both possess a good session key. It turns out that such property cannot be attained by agent A.

4.1 Modeling the Andrew Protocol in MAFSMs

As described in Section 2, a MAFSM can be constructed out of the following elements: the definition of the structure of the views; the atomic propositions of each view and how they vary over time; the choice of explicit BDI atoms of each view and how they vary over time; and the specification of the initial states for the FSMs in each view. In this section we present a MAFSM-based model of

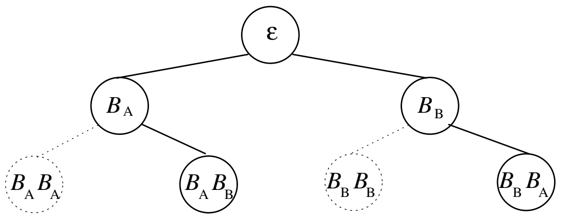


Fig. 3. The structure of views of the Andrew protocol.

the Andrew protocol, thus providing, in turn, all these elements. Since in this example we will only concentrate on beliefs of agents, in the following we will refer to BDI atoms with the name *belief atoms*.

In the MAFSM modeling the Andrew protocol there is only one FSM per view. Indeed, the processes associated to the all the views can be completely specified, starting from the protocol specification. In the presentation below, we give the FSM specifications using the input language of the model checker NuSMV.

The structure of the views. The Andrew protocol involves two agents, A and B. Therefore, we have $I = \{A, B\}$. We model each agent as having beliefs about the other. Since, for the sake of the example, we do not need to model beliefs that an agent has about itself, we will need only to consider, besides the external view ϵ, the views of A and B, the view A has about B and the view B has about A. Therefore, $OI^n = \{\epsilon, \mathbf{B}A, \mathbf{B}B, \mathbf{B}A\mathbf{B}B, \mathbf{B}B\mathbf{B}A\}$. Figure 3 shows the resulting situation. ϵ (the external observer) is modeled as a process which "sees" all the messages sent and received by the agents. $\mathbf{B}A$ and $\mathbf{B}B\mathbf{B}A$ model the behavior of agent A, while views $\mathbf{B}B$ and $\mathbf{B}A\mathbf{B}B$ model the behavior of agent B.

The set of atomic propositions P_α. In each view, we need to model an agent sending a message to another agent and/or receiving a message, as well as the properties the agent attributes to the (sub)messages it receives or sends. In particular, a (sub)message can be fresh and a key can be a good key for secret communication between the two agents. Each view has its own set of atomic propositions, reflecting the atomic properties of interest about its associated process. Since both $\mathbf{B}A$ and $\mathbf{B}B\mathbf{B}A$ model agent B (as seen from the external observer's and from A's perspective, respectively), we associate to both of them the same set of atomic propositions. Similarly for the views $\mathbf{B}B$ and $\mathbf{B}A\mathbf{B}B$. For what concerns view ϵ, following is a partial list of the atomic propositions needed:

$$P_\epsilon = \left\{ \begin{array}{l} send_{B,A}\ K'_{ab}\text{-}N'_b\text{-}K_{ab}, \\ rec_A\ K'_{ab}\text{-}N'_b\text{-}K_{ab}, \\ \dots \end{array} \right\}$$

while for what concerns the views $\mathbf{B}A$ and $\mathbf{B}A\mathbf{B}B$ in the Andrew protocol, we set:

$$P_{\mathbf{B}A} = \left\{ \begin{array}{l} \mathit{send}_B\ N_a\_K_{ab}, \\ \mathit{rec}\ N_a\_N_b\_K_{ab}, \\ \mathit{send}_B\ N_b\_K_{ab}, \\ \mathit{rec}\ K'_{ab}\_N'_b\_K_{ab}, \\ \mathit{fresh}\ N_a, \\ \mathit{fresh}\ K'_{ab}\_N'_b\_K_{ab}, \\ \mathit{shk}\ K'_{ab} \\ \ldots \end{array} \right\} \qquad P_{\mathbf{B}A\mathbf{B}B} = \left\{ \begin{array}{l} \mathit{rec}\ N_a\_K_{ab}, \\ \mathit{send}_A\ N_a\_N_b\_K_{ab}, \\ \mathit{send}_A\ K'_{ab}\_N'_b\_K_{ab}, \\ \mathit{fresh}\ N_b, \\ \mathit{fresh}\ K'_{ab}\_N'_b\_K_{ab}, \\ \mathit{shk}\ K'_{ab} \\ \ldots \end{array} \right\}$$

where the variables of the form $\mathit{rec}\,M$ or $\mathit{send}_B\,M$ (where M is a message of the Andrew protocol) are called *message variables* and represent the act of receiving M, and sending M to B, respectively. For instance, $\mathit{rec}\,N_a\_N_b\_K_{ab}$ and $\mathit{send}_B\,N_a\_K_{ab}$ in view $\mathbf{B}A$ represent A receiving $\{N_a, N_b\}_{K_{ab}}$ (message 2 in the Andrew protocol), and A sending $\{N_a\}_{K_{ab}}$ to B (message 3 in the Andrew protocol), respectively. Variables of the form $\mathit{fresh}\,M$ or $\mathit{shk}\,M$ are called *freshness variables*, and express freshness properties of (sub)messages. For instance, $\mathit{fresh}\,N_a$ in view $\mathbf{B}A$ means that N_a is believed by A to be a fresh (sub)message, while $\mathit{shk}\,K'_{ab}$ expresses the fact that K'_{ab} is a good shared key between A and B. The variables for sent messages (e.g., $\mathit{send}_{B,A}\,K'_{ab}\_N'_b\_K_{ab}$) in view ϵ are labeled (in subscript) with both the sender (B) and the intended receiver (A), while those of received messages (e.g., $\mathit{rec}_A\,K'_{ab}\_N'_b\_K_{ab}$) are labeled only with the actual receiver (A). With respect to the other views, the additional subscripts for both *send* and *rec* in the message variables in ϵ reflect the fact that the external observer, differently from the agents, knows who sends a message to whom and who receives what message.

Evolution of message variables. To specify the evolution of variables in a view we use the *next()* operator of the NuSMV input language. The *next* operator allows us to specify the next value of a variable in each state, possibly depending on its value in the current state. All variables are of type boolean, therefore the possible values are T (for true) and F (for false). Below the definitions of the next state value for some message variables in the language of view $\mathbf{B}A$ (modeling the behavior of A) is reported.[3]

```
BA
1  next(send_B N_a_K_ab) := case
                              !send_B N_a_K_ab : {T,F};
                              1 : send_B N_a_K_ab;
                            esac;
2  next(rec K'_ab_N'_b_K_ab) := case
                              send_B N_b_K_ab & !rec K'_ab_N'_b_K_ab: {T,F};
                              1 : rec K'_ab_N'_b_K_ab;
                            esac;
```

Statement 1 contains a case statement, whose first clause ($!\mathit{send}_B\,N_a\_K_{ab}$: $\{T, F\}$) contains a precondition on the left-hand side and the next value on the right-hand side. The precondition is the negation of a message variable and is

[3] Each block of statements is labeled with the name of the view $\mathbf{B}x$ in which they occur.

true when $send_B\, N_a\_K_{ab}$ is false, that is if message $N_a\_K_{ab}$ has not been sent to B yet . The next value is specified as the nondeterministic choice among the set of values $\{T, F\}$. This intuitively means that the first message of the Andrew protocol ($\{N_a\}_{K_{ab}}$) may or may not be sent (i.e., $send_B\, N_a\_K_{ab}$ may take value T or F nondeterministically) in the next state, if it has not been sent yet in the current state. The second item is the "default" clause, and it is taken if the precondition in the first clause does not apply. The result of this clause is that $send_B\, N_a\_K_{ab}$ keeps, in the next state, its current value. In Statement 2, the precondition of the first clause (i.e., $send_B\, N_b\_K_{ab}$ & $!rec\, K'_{ab}\_N'_b\_K_{ab}$) is a conjunction of two boolean expressions. The first expression means that $\{N_b\}_{K_{ab}}$ has been already sent to B, and the second means that $\{K'_{ab}, N'_b\}_{K_{ab}}$ has not been received yet. The next value again is the nondeterministic choice between values T and F (the message can be received or not). The messages in each session of the Andrew protocol are supposed to be sent and received following the order in which they occur in the protocol specification. Therefore, for each message variable, the preconditions of the next statements check whether the previous message (and therefore all the previous messages) involving the current agent (A in the case of view BA), has been already conveyed or not. The default clause is similar to that of Statement 1. The specification of the evolution of the other message variables in the other views is specified in a similar way.

Evolution of freshness variables. In any path of states of a view, freshness variables for (sub)messages originated by an agent always keep their initial values. In the Andrew protocol, this is the case for N_a in view BA (and also BBBA), as expressed by the *next* statements below, and N_b in views BB and BABB.

```
B A
3  next(fresh N_a) := fresh N_a;
```

Statement 3 simply says that *fresh* N_a keeps the current value in the next state. Similar statements need to be specified also for *fresh* N_b, *fresh* N'_b and *shk* K'_{ab} (which are messages originated by B) in the views modeling B.

On the other hand, during a protocol session, an agent can attain freshness of the messages it has not originated itself. This may happen only after it receives (possibly other) messages which contain them. Therefore, freshness variables of a message M not originated by an agent may change their value (e.g., from F to T) only when the agent has received a message containing M. After the last message of a session has been conveyed, the freshness variables of M keep their current value (no new information can be gained by the agent). Moreover, once it becomes true, a freshness variable remains stable. The above intuitions are specified as follows:

```
B A
4  next(shk K'_ab) := case
                         !shk K'_ab & !rec K'_ab_N'_b_K_ab : {T,F};
                         1 : shk K'_ab ;
                       esac;
5  next(fresh K'_ab_N'_b_K_ab) := case
                         !fresh K'_ab_N'_b_K_ab & !rec K'_ab_N'_b_K_ab : {T,F};
                         1 : fresh K'_ab_N'_b_K_ab;
                       esac;
```

Statement 4 and 5 are very similar in form. As to statement 4, the precondition in the first clause of the case statement is a conjunction of two negations ((!*shk* K'_{ab} & !*rec* K'_{ab}_N'_b_K_{ab}) meaning respectively that "K'_{ab} is not known to be a good shared key", and that the "$\{K'_{ab}, N'_b\}_{K_{ab}}$ has not been received yet". If this condition is true, the nondeterministic choice on the left-hand side of the clause is taken. The "default" clause leaves the value of the variable in the next state unchanged. Statement 5 checks, in the first clause of the case statement, if the conjunction in the precondition (!*fresh* K'_{ab}_N'_b_K_{ab} & !*rec* $K'_a b$_N'_b_K_{ab}) is true ($\{K'_{ab}, N'_b\}_{K_{ab}}$ is not known to be fresh, and it has not been received yet), and in this case nondeterministically chooses the next value of the variable.

Clearly Statements 4 and 5 are very general and do not allow us to model appropriately the freshness of (sub)messages. Indeed, additional constraints are needed. Following the BAN logic approach, there are a number of properties of messages, which relate their freshness to that of their components. For instance, an agent can conclude that a message is fresh from the freshness of one of its components. This is the case for $\{N_a\}_{K_{ab}}$, which is known to be fresh whenever N_a is known to be. NuSMV allows for specifying constraints on the admissible (reachable) states by means of invariants, which are boolean formulae that must hold in every reachable state. The following invariant statement captures some relevant constraints on some freshness variables:

BA
6 *INVAR (fresh* K'_{ab} | *fresh* N'_b*)* & *rec* K'_{ab}_N'_b_K_{ab} $\leftrightarrow$ *fresh* K'_{ab}_N'_b_K_{ab}

Invariant 6 is an equivalence ($\leftrightarrow$) whose left-hand side is a conjunction. The disjunction in the left conjunct (*fresh* K'_{ab} | *fresh* N'_b) means that K'_{ab} or N'_b is fresh; the second conjunct is meant to be true when the message $\{K'_{ab}, N'_b\}_{K_{ab}}$ has been received. Intuitively, it states that A can consider (the encrypted message) $\{K'_{ab}, N'_b\}_{K_{ab}}$ fresh (right-hand side of the equivalence) if and only if it has received the message (second conjunct on the left-hand side) and either of its components is fresh (first conjunct on the left-hand side). Similar invariants must be added for each message received by the agent.

View **B**A**B**B can be specified in a similar way. Some additional statements must be added, though, reflecting the role of agent B. Indeed, the Andrew protocol assumes that B is the key server. Therefore, B is supposed to generate and send a good shared key in message 4, whenever it believes that $\{K'_{ab}, N'_b\}_{K_{ab}}$ is fresh. Remember that we have a variable for the freshness of the last message of the protocol (namely, *fresh* K'_{ab}_N'_b_K_{ab}), and a variable for K'_{ab} being a good key (*shk* K'_{ab}). Then we can specify the above invariant as an implication:

BA**B**B
7 *INVAR fresh* K'_{ab}_N'_b_K_{ab} $\rightarrow$ *shk* K'_{ab}

Explicit belief atoms of a view. We need now to choose the explicit belief atoms of each view. In general, the choice of the appropriate set of explicit belief atoms of (the views in) a MAFSM depends on what kind of aspects of the protocol one wants to specify, and on what kind of properties need to be verified. In the case of authentication protocols, agents can only gain information carried by the messages they receive. In BAN-like logics the freshness of the received messages is based on their form, and it is a basic property to be attained by an agent. We choose, therefore, the following belief atoms as explicit beliefs atoms: the beliefs about other agents having sent or received a given message, and the beliefs about the freshness of messages. In our case, we have:

$$Expl(\mathbf{B}A, \epsilon) = \begin{Bmatrix} B_A rec\, K'_{ab}\text{-}N'_b\text{-}K_{ab}, \\ B_A fresh\, N_a \\ \ldots \end{Bmatrix} \quad Expl(\mathbf{B}B, \mathbf{B}A) = \begin{Bmatrix} B_B send_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}, \\ B_B fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab} \\ \ldots \end{Bmatrix}$$

where, for instance, $B_A fresh\, N_a$ in ϵ intuitively means that A believes that N_a is a fresh nonce; while $B_B send_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ in $\mathbf{B}A$ expresses the fact that A believes that B believes that it has sent $\{K'_{ab}, N'_b\}_{K_{ab}}$ to A. There are no explicit belief atoms in $\mathbf{B}B\mathbf{B}A$ and $\mathbf{B}A\mathbf{B}B$, as they have no beliefs atoms at all in their language. Indeed, the example we are considering does not need to model more than two nesting of the belief operators.

Evolution of explicit belief atoms. Variables representing explicit belief atoms are supposed to vary similarly to freshness variables. In particular, as long as there are still messages to be received by an agent in a view, explicit belief atoms of that view are free to change value from F to T1. Once the last message of the protocol has been received, no new belief can be attained, and explicit belief atoms keep their value, thereafter. Again, once they become true, they remain stable. The following statement specifies how the value of $B_B send_A\, N_a\text{-}N_b\text{-}K_{ab}$ may vary in $\mathbf{B}A$:

```
BA
8  next($B_B send_A\, N_a\text{-}N_b\text{-}K_{ab}$) := case
                                   !$B_B send_A\, N_a\text{-}N_b\text{-}K_{ab}$ & !rec $K'_{ab}\text{-}N'_b\text{-}K_{ab}$ : {T,F};
                                   1 : $B_B send_A\, N_a\text{-}N_b\text{-}K_{ab}$;
                                 esac;
```

Very similar statements must be added for all the explicit beliefs in all the views. Similarly to freshness variables, additional constraints, in the form of state invariants, on reachable states need to be added for explicit beliefs atoms. In particular we report below some relevant ones for the Andrew protocol. All these invariants can be seen as encodings of standard properties holding in BAN-like logics.

```
BA
9  INVAR rec $K'_{ab}\text{-}N'_b\text{-}K_{ab}$ → $B_B send_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$
10 INVAR fresh $K'_{ab}\text{-}N'_b\text{-}K_{ab}$ → $B_B fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$
```

Both invariants are implications. Intuitively, Invariant 9 states that if it receives the message $\{K'_{ab}, N'_b\}_{K_{ab}}$ (left-had side of the implication) then A ascribes to B the belief that B has sent that message to A (right-hand side of the implication). Invariant 10 states that if it can conclude that $\{K'_{ab}, N'_b\}_{K_{ab}}$ is fresh, then A also ascribes to B the same belief.

Initial states of a view. Finally, we have to set the initial states of the FSM in each view. They can be specified by a boolean formula which identifies all and only the admissible initial states of the FSM. A protocol session starts with no

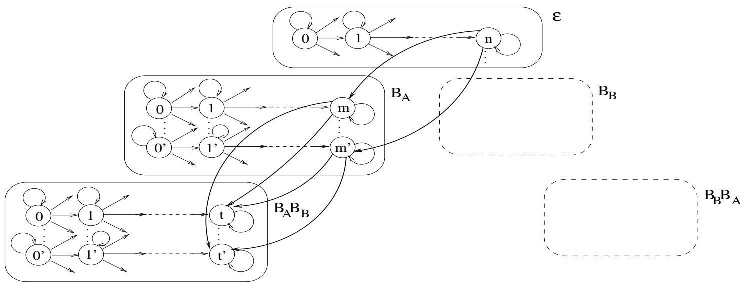


Fig. 4. The MAFSM for the Andrew protocol.

message sent or received. Thus, the process in each view starts with the value of all the message variables set to false. Since no message has been received yet, freshness variables of messages not originated by an agent are set to false as well. All the other variables can take nondeterministically the value T or F in the initial states.

4.2 Model Checking the Andrew Protocol

In this section, we show how the model checking algorithm described in Section 3 works, trying to check a desired property of the Andrew protocol. Let us consider the following property: A believes that B believes that K'_{ab} is a good shared key, any time it receives the last message of the Andrew protocol (see [3]). This property can be written as the following MATL formula in view ϵ:

$$\epsilon : \mathsf{AG}\,(rec_A\,K'_{ab}\text{-}N'_b\text{-}K_{ab} \wedge B_A\,fresh\,N_a \rightarrow B_A\;B_B\,shk\,K'_{ab}) \tag{1}$$

where the (sub)formula $B_A\;B_B\,shk\,K'_{ab}$ is an implicit belief atom.

Phase A and B. We only have to consider the belief operator O_A. We construct the set $ArgImpl(\mathbf{B}B, \epsilon, \Gamma) = \{B_B shk K'_{ab}\}$. Since $B_B\,shk\,K'_{ab}$ is an implicit belief atom, MAMC-View $(\epsilon, \{\mathsf{AG}(rec_A K'_{ab}\text{-}N'_b\text{-}K_{ab} \wedge B_A fresh N_a \rightarrow B_A\;B_B\,shk\,K'_{ab})\})$ calls itself recursively on view $\mathbf{B}A$ and $\{B_B\,shk\,K'_{ab}\}$ (performing a call to the algorithm MAMC-View($\mathbf{B}A$, $\{B_B\,shk\,K'_{ab}\}$)). In view $\mathbf{B}A$, we need to consider the operator O_B and to compute $ArgImpl(\mathbf{B}B, \mathbf{B}A, \{B_B\,shk\,K'_{ab}\}) = \{shk\,K'_{ab}\}$. MAMC$-$View($\mathbf{B}A$, $\{B_B\,shk\,K'_{ab}\}$) descends to view $\mathbf{B}A\mathbf{B}B$ (performing a call to MAMC$-$View($\mathbf{B}A\mathbf{B}B, \{shk\,K'_{ab}\}$)). Phase B is not performed in view $\mathbf{B}A\mathbf{B}B$ as no implicit beliefs occur in the input formulae (namely, $\{shk\,K'_{ab}\}$).

Phases C.1 and C.2 at $\mathbf{B}A\mathbf{B}B$. The only formula to check in this view is the atomic formula $shk\,K'_{ab}$. The FSM of this view already contains the information about its truth value in each state (see Section 4.1). Therefore, both phases end immediately.

Phase C.1 at $\mathbf{B}A$. The FSM f of this view is labeled with the implicit belief atom $B_B\,shk\,K'_{ab}$. For each reachable state s of f and each pair $\langle f', s'\rangle$ satisfying the formulae in the set $ArgExpl(\textsc{b}B, \textsc{b}A, s)$ (i.e. the pairs compatible with state s) the intersection of $L(s')$ and the set of arguments of implicit belief atoms $ArgImpl(\textsc{b}B, \textsc{b}A, \{B_B\, shk\, K'_{ab}\}) = \{shk\, K'_{ab}\}$ is computed. This gives either the empty set (meaning that $shk\, K'_{ab}$ is not true in s') or $\{shk\, K'_{ab}\}$ itself (meaning that $shk\, K'_{ab}$ is true in s'). The final step consists in adding to $L(s)$ the implicit belief atom $B_B\, shk\, K'_{ab}$, if every state of f', compatible with s, satisfies $shk\, K'_{ab}$. It turns out that the states of $\textsc{b}A$ satisfying that implicit belief are those which satisfy $rec\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ (A has received message 4) and $fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ (message 4 is known to A to be fresh). All those states also satisfy the explicit belief atom $B_B\, fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ (by Invariant 10), and are therefore compatible only with those states of view $\textsc{b}A\textsc{b}B$ where $fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ is true. Notice that there can be a reachable state of $\textsc{b}A$ where $rec\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ and $fresh\, N_a$ are both true but $fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ is false. This is possible, as nothing in message 4 is recognizable by A as fresh. As a consequence, there is a reachable state of view $\textsc{b}A$, which does not satisfy $B_B\, fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ either. Let the state m of view $\textsc{b}A$ in Figure 4 be one such state. Therefore, state m satisfies $rec\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ and $fresh\, N_a$ but not $B_B\, shk\, K'_{ab}$.

Phase C.2 at view $\textsc{b}A$. Once again, the formula to check is atomic (though an implicit belief atom). Therefore, this phase ends immediately.

Phase C.1 at ϵ. We have now to process the implicit belief $B_A\, B_B\, shk\, K'_{ab}$. It performs similar steps as in phase C.1 for view $\textsc{b}A$. It turns out that there is (at least) a reachable state in the FSM of ϵ which does not satisfy this implicit belief. Indeed, as we have pointed out above, there is a state of view $\textsc{b}A$ (state m in Figure 4) which does not satisfy $B_B\, shk\, K'_{ab}$ but satisfies both $fresh\, N_a$ and $rec\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$. Let us assume that state n in ϵ (see Figure 4) satisfies $B_A\, fresh\, N_a$ and $rec_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$ but does not satisfy the explicit belief atom $B_A\, fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$. n is actually a reachable state of the FSM of ϵ. Since n satisfies $rec_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$, by an invariant of ϵ similar to Invariant 9, n also satisfies $B_A\, rec_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$. Since n does not satisfy the belief atom $B_A\, fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$, the explicit belief atoms true at n are not enough to rule out the compatibility with state m in $\textsc{b}A$ (see again Figure 4). Indeed, m does not satisfy $fresh\, K'_{ab}\text{-}N'_b\text{-}K_{ab}$, but it satisfies the formulae in the set $ArgExpl(\textsc{b}A, \epsilon, n)$ (the arguments of the explicit belief atoms true at n), and is therefore compatible with state n. State m, as we know, does not satisfy $B_B\, shk\, K'_{ab}$. As a consequence of the notion of satisfiability of implicit BDI atoms (Section 2), state n of ϵ does not satisfy $B_A\, B_B\, shk\, K'_{ab}$.

Phase C.2 at ϵ. Here the usual CTL model checking algorithm on the formula (1) is called. As expected, the final answer is negative since, from phase C.1 in ϵ, there are reachable states satisfying $rec_A\, K'_{ab}\text{-}N'_b\text{-}K_{ab} \wedge B_A\, fresh\, N_a$ but not $B_A\, B_B\, shk\, K'_{ab}$.

4.3 Analysis of the Andrew Protocol Corrected

In the previous section we presented and checked a model of the original version of the Andrew protocol. That version was known to be buggy. [3] proposed a corrected version of the protocol which does not suffer form the same problem. Indeed, as we already pointed out, they noticed that the problem is just that in the last message there's nothing that A can recognize as fresh.

The fix proposed is very simple, but effective in eliminating the bug. Indeed it's sufficient to concatenate the nonce N_a to the new key and the new nonce. Remember that B has received N_a and N_a is known to A to be fresh. Therefore, the protocol is rewritten by rewriting the forth message of the protocol as follows:

$$4' \;\; B \rightarrow A \;:\;\; \{K'_{ab}, N'_b, N_a\}_{K_{ab}}$$

In order to build a model of this new version of the protocol, we need to reflect the simple modification above into the model constructed in the previous section. This corresponds to substituting in all the views each occurrence in a message variable or explicit belief of the string $K'_{ab}\_N'_b\_K_{ab}$ with $K'_{ab}\_N'_b\_N_a\_K_{ab}$. This change in the syntax of the variables used to specify the protocol, also induce some obvious changes in the constraints the corresponding variables have to comply to. In particular, we would need to slightly modify invariant 6 as follows:

BA
6' *INVAR (fresh* K'_{ab} | *fresh* N'_b | *fresh* N_a*) & rec* $K'_{ab}\_N'_b\_N_a\_K_{ab}$ $\leftrightarrow$ *fresh* $K'_{ab}\_N'_b$ $\_N_a\_K_{ab}$

reflecting the fact that, since there is another element in message 4, now such element contributes to the freshness of the entire message. All other invariants (except for the substitution of each occurrence of message 4 with its new version) remain unchanged.

Interestingly enough, this simple change in the model, actually allows the model checking algorithm to answer positively to the check of the property we are interested in. The new property is now:

$$\epsilon : \mathsf{AG}\,(rec_A\, K'_{ab}\_N'_b\_N_a\_K_{ab} \wedge B_A\, fresh\, N_a \rightarrow B_A\;\, B_B\; shk\; K'_{ab}) \qquad (2)$$

In verifying 2, MAMC-View goes through the same steps as in the previous case. The difference is that now, when the algorithm performs **Phase C.1 at view** BA, all the reachable states satisfying $rec\,\mathrm{K}'_{ab}\_\mathrm{N}'_b\_\mathrm{N}_a\_\mathrm{K}_{ab}$ and $fresh\,N_a$ need also to satisfy $fresh\,K'_{ab}\_\mathrm{N}'_b\;\_N_a\_\mathrm{K}_{ab}$, by the new Invariant $6'$. By the analogue of Invariant 10, all the reachable states also satisfy $B_B fresh\,\mathrm{K}'_{ab}\_\mathrm{N}'_b$ $\_N_a\_\mathrm{K}_{ab}$. Therefore, all such reachable states are compatible with the states in view BABB satisfying $fresh\,\mathrm{K}'_{ab}\_\mathrm{N}'_b\;\_N_a\_\mathrm{K}_{ab}$, an+d, by the analogue of Invariant 7, the compatible states in view BABB must satisfy $shk\,\mathrm{K}'_{ab}$. With respect to **Phase C.1 at** BA as described in the previous section, this means that no state m of Figure 4 is reachable anymore. Only the states m' in figure, will be reachable, and those are the state which are compatible with states of BABB satisfying $shk\,\mathrm{K}'_{ab}$. As a consequence, after this phase, all the reachable states of BA satisfying both $rec\,K'_{ab}\_\mathrm{N}'_b\_\mathrm{N}_a\_\mathrm{K}_{ab}$ and $fresh\,N_a$, also satisfy $B_B shk\,\mathrm{K}'_{ab}$.

By a similar reasoning, when performing **Phase C.1 at view** ϵ, the algorithm discovers that all the reachable states satisfying both $rec_A\,K'_{ab}\_N'_b\_N_a\_K_{ab}$ and $B_A\,fresh\,N_a$, must be compatible with states of BA satisfying $B_B shk\,\mathrm{K}'_{ab}$. Now the new Invariant $6'$ rules out compatibility of the states n (see Figure 4) with any state of $\textsc{b}A$ (recall that the state m is not reachable anymore in view $\textsc{b}A$). Therefore also those states n vacuously satisfy the conclusion.

Finally, **Phase C.2 at view** ϵ, while performing CTL model checking, finds out that property (2) is now satisfied, as expected.

5 Conclusion

We have described a model-checking based verification procedure for multiagent systems, and shown how it can be effectively employed in the analysis of multiagent systems. To substantiate this claim, we have formalized within our framework a multiagent protocol, namely the well known Andrew protocol, of which we were interested in analyzing some security properties. The protocol has been modeled and analyzed both in its original (faulty) version and the corrected one.

References

1. M. Abadi and M. Tuttle. A semantics for a logic of authentication. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 201–216, 1991.
2. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. *Journal of Logic and Computation, Special Issue on Computational & Logical Aspects of Multi-Agent Systems*, 8(3):401–423, 1998. Also IRST-Technical Report 9708-07, IRST, Trento, Italy.
3. M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'99)Trento, Italy. July 1999.*
5. E. Clarke, O. Grumberg, and D. Long. Model Checking. In *Proceedings of the International Summer School on Deductive Program Design*, Marktoberdorf, Germany, 1994.
6. E. Giunchiglia and F. Giunchiglia. Ideal and Real Belief about Belief. In *Practical Reasoning, International Conference on Formal and Applied Practical Reasoning, FAPR'96*, number 1085 in Lecture Notes in Artificial Intelligence, pages 261–275. Springer Verlag, 1996.
7. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
8. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principle of Knowledge Representation and Reasoning.* Morgan Kaufmann Publishers, 1991.

Towards Formal Specification and Verification in Cyberspace*

Stanislaw Ambroszkiewicz^{1,2}, Wojciech Penczek^{1,2}, and Tomasz Nowak^{1}

^{1} Institute of Computer Science
Polish Academy of Sciences, Warsaw, Poland
^{2} Akademia Podlaska
Institute of Informatics
Siedlce, Poland
{penczek, tnowak, sambrosz}@ipipan.waw.pl

Abstract. A formal framework for specification and verification of multi-agent systems is developed. Specification of an infrastructure created by a mobile agent platform is presented. On the basis of the specification, the notions of common ontology core, and agent's knowledge are introduced. A simple agent architecture is presented. Given agents' knowledge and decision mechanisms, model checking method is applied to verify whether the agents can realize their goals.

1 Introduction

Computer networks offer new application scenarios that cannot be realized on a single workstation. However, for large computer networks like the cyberspace (the open world created by the global information infrastructure and facilitated by the Internet and the Web) the classical programming paradigm is not sufficient. It is hard to imagine and even harder to realize the control, synchronization and cooperation of hundreds of processes running on remote hosts.

The idea of mobile software agents inhabiting the cyberspace seems to be a good solution here. A mobile agent is an executing program that can migrate from host to host across a heterogeneous network under its own control and interact with other agents. However, it is clear that a single agent cannot perform efficiently its tasks in a large open world without cooperation with other agents.

For large open worlds simple cooperation mechanisms based on bilateral communication are not sufficient. Sophisticated interaction mechanisms and services are needed. During the interactions agents communicate, negotiate, and form organization structures. To realize the concept of mobile agents together with agents interactions and service infrastructure, a special middleware called "mobile agent platform" (MAP, for short) is needed. There is a number of platforms available over the Internet, for example IBM Aglets, Concordia, Grasshopper, Mole, and Voyager to mention only some of them. One of them is Pegaz [15]

* Partly supported by the State Committee for Scientific Research under the grants No. 7 T11C 040 20, 7 T11C 006 20, and Esprit under the grant No. 20288 CRIT-2.

developed at our Institute. Platforms serve for creating infrastructures on computer networks, see Fig 1., so that details of network functioning are hidden from the users as well as from the agents. This makes programming more easy. A programmer need not manually construct agent communication, nor agent transportation. This may be viewed as a high level abstraction from network communication protocols, operation systems (due to Java), and data structures.

Most of the existing mobile agent platforms create similar infrastructures. However, there is a problem of interoperability between different platforms if agents are to migrate from one platform to another. Based on OMG CORBA, MASIF standard [13] offers interoperability limited to implementation independent aspects. Agents that migrate from platform A to platform B must know how to access the internal services and resources offered by the platform B. Such object-internal aspects are usually not handled by CORBA.

Since most of today's mobile agent platforms are implemented in Java, it is postulated in [11] to create a new OMG standard to define implementation-specific (i.e. Java-specific) convention that allows a mobile agent to migrate to any standard-compilant platform. This concept (a specification of a generic mobile agent platform) has been developed in GMD FOCUS and IKV++ [5], and realized in the Grasshopper platform [6]. However, the interoperability limited only to the core functionality offered by the Grasshopper architecture is not sufficient for a mobile agent to be able to access internal services and resources (of a remote place) being beyond the scope of the core functionality. To do so the agent must "know" how and for what purpose the services and resources can be used, or at least to be able to learn about that. In order to assure this, a common language with machine processable semantics is needed. The language is also necessary for defining agent negotiation protocols, joint plans, and intentions. Usually semantics is expressed by an ontology of the language domain. According to the standard definition, an ontology is an explicit specification of a conceptualization, see [7]. A conceptualization is an abstract, simplified view of the world. Hence, in order to define an ontology and a language, a formal representation of the world is needed. In our case, the world is the environment (called cyberspace) created by mobile agent platforms. Since most of the existing MAPs create similar infrastructures, it seems that a formal specification of the cyberspace is possible. In our opinion the common ontology core based on a formal specification of the cyberspace is a way to achieve a high interoperability level.

There is an effort taken by FIPA [4]. This approach to ontology is based on a different point of view. It is supposed that each domain (for example, e-commerce) has its own specification expressed in a formal language. This specification is identified with the ontology of the domain. So that the ontology provides a vocabulary for representing and communicating knowledge about the domain. If two agents wish to converse, they must share a common ontology for the domain of discourse. Other proposals try to set standards for describing and exchange ontologies. These standards must include appropriate modeling primitives for representing ontologies, define their semantics and appropriate syntax for representing ontologies. These proposals are: DARPA Agent Markup Language (DAML) [3], the Ontology Interchange Language (OIL) [17], OntoBroker [18] to mention only few of them.

Our approach is a bit different. In Section 2. we construct a representation of cyberspace as the basis for constructing ontology core for application domain. This representation serves also as the basis for constructing the key aspects of agent architecture (Section 3.), i.e., agent knowledge, perception mechanisms, and a simple agent communication language. Our approach to semantics follows the idea of Wittgenstein [27] that the meaning of language is in its use, whereas in the approach of Gruber et al. [7] and Guarino [8], the interpretation of concepts is constrained by logical axioms. Equipping agents with goals and decision making mechanisms, we construct multi-agent systems. In Section 4. we develop a formal theory (based on modal logic) of such multi-agent systems. Model checking algorithm of the formulas of our language is presented and its complexity is discussed.

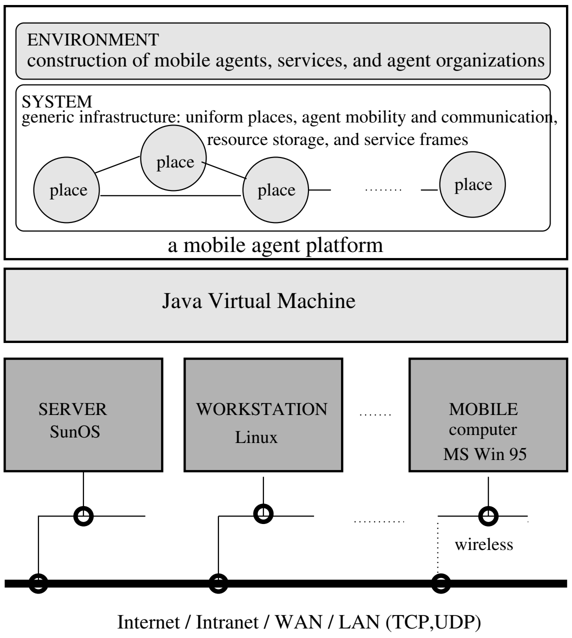

ENVIRONMENT
construction of mobile agents, services, and agent organizations
SYSTEM
generic infrastructure: uniform places, agent mobility and communication, resource storage, and service frames
place
place
place
place
a mobile agent platform
Java Virtual Machine
SERVER SunOS
WORKSTATION Linux
MOBILE computer MS Win 95
wireless
Internet / Intranet / WAN / LAN (TCP,UDP)

Fig. 1. Functioning of a mobile agent platform, e.g. Pegaz

2 Generic MAP Environment

The main components of the infrastructure (created by a platform like Grasshopper or Pegaz) are places, services located at the places, and agents that can move from place to place. Agents use the services, communicate, and exchange data and resources with other agents. They can also collect, store, exchange, and transport the resources.

In our formal description the primitives are: places, agents, actions, services, and resources. Places are composed into a graph. An edge between two nodes in the graph expresses the fact that there is an immediate connection between the corresponding places. Resources are primitives grouped into types, analogously as types of data, and objects in programming languages. Services are represented as operations on resources and are of the form: $Op : C \rightarrow B$, producing resource of type B from resource of type C. Primitive types of resources and services are specific for the application domain.

The infrastructure is developed by constructing sophisticated services (i.e., enterprises, organizations) by the agents. To do so, the agents should be equipped with appropriate mechanisms that are called *routines.* Routines are composed of the following primitive actions: **move** to another place, **use a service**, **agent cloning** (i.e. agent creates a copy of itself), **get a resource** from other agent, **give a resource** to other agent, **communicate** with other agent, **finishing agent activity**.

Let A denote the set of agents' primitive actions. Some actions belonging to the core functionality of a MAP are nor considered here like agent naming, registration, and security. It is a subject of discussion whether the set of primitive actions proposed above is comprehensive enough to define all the actions that can be useful for the agents.

Joint action a (for example a communication action) can be executed if for all the agents needed for an execution, the action a is enabled and selected for execution, i.e., intuitively all the agents "can" and "do want" to participate in the execution of this action. If one of the agents cannot or doesn't want to participate in the execution, then the attempt to execute action a fails. For this reason we assume that for any joint action a, and any agent i, needed to execute a, there is a local action $fail(a, i)$ of agent i that corresponds to agent i's failure to execute joint action a.

The crucial notion needed to define a representation of the "world", presented intuitively above, is the notion of "event". An event corresponds to an action execution, so that it "describes" an occurrence of a local interaction of the agents participating in the execution. The term "local" is of great importance here. Usually, local interaction concerns only a few agents so that the event associated with this local interaction describes only the involved agents, services, and the interaction places. The events form a structure, which expresses their causal relations.

Having primitive concepts and setting initial conditions for agents, places, and distribution of primitive resources and services, the set E of all events that can occur in the environment can be defined.

3 Overview of Agent Architecture

In order to construct efficient agents, their architecture must be simple. Therefore, we need a simple, but still flexible, notion of knowledge that can be imple-

mented into agent architecture. Hence, we face the following problems:
How to represent, store, pass, and reason about knowledge in an efficient way, so that it can be implemented into simple agent architecture ?

3.1 Knowledge and Language

Classical definitions of knowledge are built on global states and global time, see Halpern et al. [9]. The consequence of that definition is logical omniscience of the agents. This very omniscience is frequently regarded as a drawback especially if an agent is to take decisions in real time. Moreover, when modeling such an agent, it turns out that the representation of the whole world must to be put into the "brain" of the agent, see the concept of BDI-agent of Rao & Georgeff [25]. This is acceptable if the world is small, say up to a few agents, but if the world is getting larger, then it is computationally unrealistic do deal with such a model, see [2]. Hence, if the world is large and/or what is even worse, the world is "open," then the classical notion of knowledge remains only an elegant theoretical notion.

Our alternative proposal to the classical notion of knowledge consists in assuming that initially agents know almost nothing and acquire knowledge during local interactions by perception and communication.

We assume that knowledge of each agent i can be stored in the variables $v^i_1, \ldots, v^i_K$ ranging over the sets $W_1, \ldots, W_K$. The sets W_k can be of any sort including: char, integers, reals, etc.

This variable collection defines **agent knowledge frame,** It may be seen as relational database abstraction. It is important to note that the range of variables v^i_k of agent i and v^j_k of another agent j is the same, that is, W_k. So that, any variable stands for a common concept for all the agents in the system.

The profile of current values of agent's variables (database) is considered as the current agent's mental state. All possible agents' mental states are collected in the set $M = \prod_{k \in K} W_k$. For example, $m_i = (w_1, w_2, \ldots, w_K)$ corresponds to agent i's mental state where $v^i_1 = w_1, v^i_2 = w_2, \ldots, v^i_K = w_K$.

Values of the variables $(v^i_1, \ldots, v^i_k)$ are updated by agent i's perception and revised by knowledge acquired from other agents via explicit communication.

Next, we define agent perception. Let $P = \{p_1, p_2, ..., p_l\}$ be a set of common observables (propositions of the Propositional Calculus) of all the agents. Usually, the propositions characterize agent locations, their resources, services available on places, and so on. The propositions are evaluated at events. The value of a proposition at an event is either true or false, or undefined. Since each event is local for the agents participating in its execution, values of some propositions (describing agents or places not participating in the execution) cannot be determined and are left undefined.

The propositions are evaluated at events in the following way: at event e any agent participating in this event can evaluate any $p \in P$ and come up with the conclusion whether p is true or false or unknown. The value "unknown" refers to the propositions with undefined value at this event as well as to the propositions that cannot be evaluated by the agent because they are hidden for the agent, for example propositions concerning the storage contents of another agent participating in event e.

Let $O = \{t, *, f\}^l$ (l is the number of propositions in the set P) be the set of all possible strings of length l over the set $\{t, *, f\}$ coding all the valuations of the propositions. For example, for $l = 5$, the string $o = ttf * t \in O$ corresponds to agent's observation that propositions p_1, p_2 and p_5 are true, proposition p_3 is false, whereas the value of proposition p_4 is unknown. The set O is the collection of all possible observations.

Agent i's perception is defined as a function $\Pi_i : E_i \longrightarrow O$. An intuitive meaning of $\Pi_i(e) = ttf * t$ is that at event e agent i perceives the propositions in the way described above.

Common updating mechanism is defined as a function $Mem : M \times O \longrightarrow M$. An intuitive meaning of $Mem(m, o) = m'$ is that mental state m is updated to m' after observation o has been made by an agent.

Let $Q = \{*, ?\}^K$ be the set of *queries.* The interpretation of query $q = ***?*?$ is: **tell me the value of variable v_4 and v_6 in your memory (database)**. The set of all possible answers to the queries Q is defined as follows. $Val(Q) \overset{df}{=} \prod_{k \in K}(W_k \cup \{*\})$.

Let query $q = q_1 \ldots q_K$ be sent to agent i, whose memory state is $m_i = (w_1, \ \ldots \ , w_K)$. The answer of agent i is the string $(x_1 \ldots x_K)$, where $x_j = w_j$, for $q_j =?$, and $x_j = *$ for $q_j = *$. For example, agent i's answer to query $* * *?*?$ can be $* * *2 * 9$.

In the set of actions A we identify a subset of *communication actions.* For each query $q \in Q$ we have a joint action iqj of agents i and j with the intended meaning of sending query q to agent j by agent i. If agent j does agree to answer the query, then the action iqj is successfully executed, otherwise it fails.

Common revision mechanism is defined as a function: $\Xi : M \times Val(Q) \longrightarrow M$. An intuitive meaning of $\Xi(m, x_1 \ldots x_K) = m'$ is that mental state m is revised to m' after receiving answer $x_1 \ldots x_K$ to its query sent to another agent.

Agent common ontology core is defined as $\Lambda = (E, M, O, Q, Mem, \Xi)$. The core consists of the set of events E, the set of mental states M, the set of observations O, the set of queries Q, and two "reasoning" mechanisms Mem and Ξ. The first one for updating agent mental state if a new observation has been made by the agent. The second one for revision of the current mental state if the agent has received an answer to its query. These two mechanisms express the meaning of agent primitive concepts.

The concept defined above is merely a frame of ontology core rather than a fully specified ontology. The frame is built (as a component) into our agent architecture to be presented in the next section. It is clear that an ontology core specification should be comprehensive enough to capture all essential concepts that can be useful for the agents. It is a subject of testing and verification on simple applications how to construct useful concepts from the primitive ones introduced in the specification of generic MAP environment in Section 2.

When the knowledge is exchanged between agents the following updating problem occurs. Suppose that an agents received via communication from another agent a different information that he posses on the same subject. The agent must decide which one is the most recent one. The way of solving this kind of problems is by encoding an additional information, which is sufficient for deciding which agent has got the latest information about the subject. One of the possible solutions is so called secondary information defined for gossip automata [14].

3.2 Semantic Interoperability

It is assumed that the frame of ontology core defined above is common for all the agents. Hence, all the agents have the same primitive concepts of generic MAP environment, and do use these primitive concepts in the same way. The simplest way to achieve semantic interoperability is to assume that all agents have the same concepts (not only the primitive ones) and use them in the same way. This makes passing knowledge (via queries) from one agent to another meaningful. However, even then the agents are not identical. They are differentiated by their perceptions, and histories, i.e., by what they have observed and learned from other agents in the past.

The cyberspace is a heterogeneous environment, so that the assumption that all agents have the same concepts with the same meaning is too restrictive. Users who construct agents and services may built new compound concepts into agent and service architecture. They can do so, however in order to assure semantic interoperability, they must specify formally (in a machine readable-way):

– How are the new introduced concepts constructed from the primitive and already constructed concepts?
– How are they used, i.e. modified by already constructed concepts, and by knowledge acquired from perception and communication?

Now, it is clear that the primitive concepts as well as rules of using primitive and compound concepts are crucial for achieving semantic interoperability.

3.3 Decision Making Mechanism

Agent's local state (denoted by l) is composed of a mental state m and contents w of its own storage. Contents of agent's storage is determined by the resources the agent has got. Let W denote the set of possible contents of agent's storage and M be the set of all possible agent's mental states. Define $L \overset{df}{=} M \times W$ to be the set of all possible agent's local states. Let $Z = \{z_1, z_2, ..., z_m\}$ be a collection of some subsets of L. Each z_i is called a *situation*. The choice of the set Z should be determined by a specific application of the agent architecture. The situations may be viewed as high level concepts used by the agent in its decision making.

Next, define $Z(l) \overset{df}{=} (c_1, c_2, ..., c_m) \in \{0,1\}^m$ such that $c_i = 1$ if $l \in z_i$, otherwise $c_i = 0$. Let $Sign \overset{df}{=} \{Z(l) : \quad l \in L\}$ be called the set of *signals* for the agent to react. Actually, the signals can be represented by formulas $\phi_1, \phi_2, ..., \phi_m$

(expressed in the modal language defined in Section 4.1) evaluated at agent's local states such that ϕ_i is true at l if and only if $l \in z_i$.

Additionally each agent i has goal g_i to fulfill. The goal is represented by a subset of situations such that the agent wants to reach one of them, i.e., $g_i \subseteq Z$. Let $G_i \subseteq 2^Z$ denote the set of possible agent i's goals. It is assumed that to realize its goal, each agent i has a library of available routines *Routines* at its disposal. Each routine is of the form $routine \overset{df}{=} (z, f, z', prob, cost)$, where z, z' are situations and $f : Sign \longrightarrow 2^{A_i}$ is a knowledge-based protocol, where A_i is the set of actions of agent i and z, z' correspond to the pre and post conditions of the protocol. It is supposed that if the routine is applied by the agent, it leads in a finite number of steps from situation z to situation z' with probability *prob* and cost *cost*. That is, for any $l \in z$ and for any sequence of local states $(l_0, l_1, l_2, ..., l_k, ...)$ such that $l_0 = l$ and l_k is a result of performing an action from the set $f(l_{k-1})$, there is n such that $l_n \in z'$.

Agent i's decision mechanism is defined as a function: $Dec_i : G_i \times Sign \longrightarrow Routines$. Suppose that the current agent i's goal is g_i and its local state is l. Then, $Dec_i(g_i, Z(l)) = r$ means that in order to realize the goal g_i the agent chooses the routine r at the signals given by $Z(l)$. Using its decision mechanism agent i can determine a path leading from its current situation, say z, to one of its goal situations, for example $z \overset{f_1}{\to} z_1 \overset{f_2}{\to} z_2 \overset{f_3}{\to} \ ... \ \overset{f_n}{\to} z_n \in g_i$.

If an agent is supposed to be rational, then the decision mechanism can not be arbitrary, e.g., it may be Bayesian mechanism [1] that consists in choosing the routines that minimize the expected cost of getting to a goal situation.

4 Formal Theory of MAS

Our formal theory of Multi Agent System (MAS) uses an event-based approach. So, let N be a finite number of agents, $E = \bigcup_{i=1}^{N} E_i$ - a set of events, where E_i is a set of events agent i participates in, for $1 \leq i \leq N$, and $agent(e) = \{i \in N \mid e \in E_i\}$. Event e is called *joint* if belongs to at least two different sets E_i. Event structures have been successfully applied in the theory of distributed systems [26] and several temporal logics have adopted them as frames [10,16,19,20,21, 22]. Next, we present a formal definition of event structure.

Definition 1. *A* **labelled prime event structure** *(***lpes***, for short) is a 5-tuple $\mathcal{ES} = (E, A, \to, \#, l)$, where*

1. *E is a finite set, called a set of* events *or* action occurrences*,*
2. *A is a finite set, called a set of* actions*,*
3. *$\to \subseteq E \times E$ is an irreflexive, acyclic relation, called the* immediate causality relation between the events *such that $\downarrow e \overset{def}{=} \{e' \in E \mid e' \to^* e\}$ is finite for each $e \in E$, where $\to^*$ is the reflexive and transitive closure of $\to$,*
4. *$\# \subseteq E \times E$ is a symmetric, irreflexive relation, called* conflict relation*, such that $\# \circ \to^* \subseteq \#$ (called* conflict preservation*),*
5. *$l : E \longrightarrow A$ is a labelling function.*

Two events e, e', which are not in $\rightarrow^*$ nor in $\#$ are concurrent, denoted $e \| e'$.

The conflict preservation condition specifies that the conflict of two events is inherited by all the events in their causal futures. The labeling function l indicates for each event which action is executed, i.e., $l(e) = a$ means that event e is an occurrence of action a. As with events each action belongs to a set of agents, denoted $agent(a) = \{i \in N \mid a \in l(E_i)\}$. The two relations $\rightarrow$ and $\#$ capture the causality and conflict relationship between events, respectively. Two events e, e' are in immediate causality relation, i.e., $e \rightarrow e'$ if the same agent participates in them, and the agent has executed the action $l(e')$ immediately after the execution of action $l(e)$. Two events are in conflict if they cannot occur in the same run. One of the sources of conflict is a choice of actions.

Let $E_N \overset{def}{=} \{(e, i) \in E \times N \mid i \in agent(e)\}$ denote the set of *local state occurrences* (lso's, for short), i.e., (e, i) represents the lso of agent i reached after executing event e. Since our language is to be interpreted over lso's rather than over events, so for each lpes we define the corresponding lso-structure.

Definition 2. *Let $\mathcal{ES} = (E, A, \rightarrow', \#', l')$ be an lpes. The* **lso structure** *corresponding to $\mathcal{ES}$ is defined as $\mathcal{S} = (E_N, A, \rightarrow, \#, l)$, where*

1. *$(e, i) \rightarrow (e', j)$ iff $e \rightarrow' e'$ and $i \in agent(e')$,*
2. *$(e, i) \# (e', j)$ iff $e \#' e'$,*
3. *$l : E_N \longrightarrow A$ such that $l(e, i) = l'(e)$, for $e \in E$.*

Two lso's s, s', which are not in $\rightarrow^*$ nor in $\#$ are concurrent, denoted $s \| s'$.

The above definition needs some explanation. Intuitively, for two lso's $(e, i) \rightarrow (e', j)$ if (e, i) is one of the starting lso's of event e'. Intuitively, two lso's are concurrent if they can be reached by the system at the same time. According to the definition two lso's $(e, i), (e', j)$ are in the relation $\|$ if they correspond either to the same event (then $e = e'$) or to concurrent events, or to causally related events $(e, i), (e', j)$, which are not comparable by $\rightarrow^*$. Notice that $e \rightarrow'^* e'$ iff $(\exists k, j \in N) : (e, k) \rightarrow^* (e', j))$.

Consider the lso structure corresponding to the two synchronizing agent system represented by the Petri Net in Figure 2. We have added two starting lso's corresponding to an artificial action @ putting tokens to the places 1 and 2. The agents can synchronize by executing the joint action e. The immediate causality relation is marked by the arrows, the concurrency relation by the dotted lines, whereas the conflict relation is not marked.

Our present aim is to incorporate into the formalism the notions of agent's **knowledge**, **goals** and **intentions**. The first step towards this aim is to define runs of lpes's. A run plays a role of "reality" (see [24]), i.e., it describes one of the possible "real" full behaviours of the agents.

Definition 3. *Let $\mathcal{S}$ be an lso-structure. A maximal, conflict-free substructure $(R, A, \rightarrow_r, \emptyset, l)$ of $\mathcal{S}$ is a* **run**. *The set of all runs of the lso-structure is denoted by $\mathcal{R}$.*

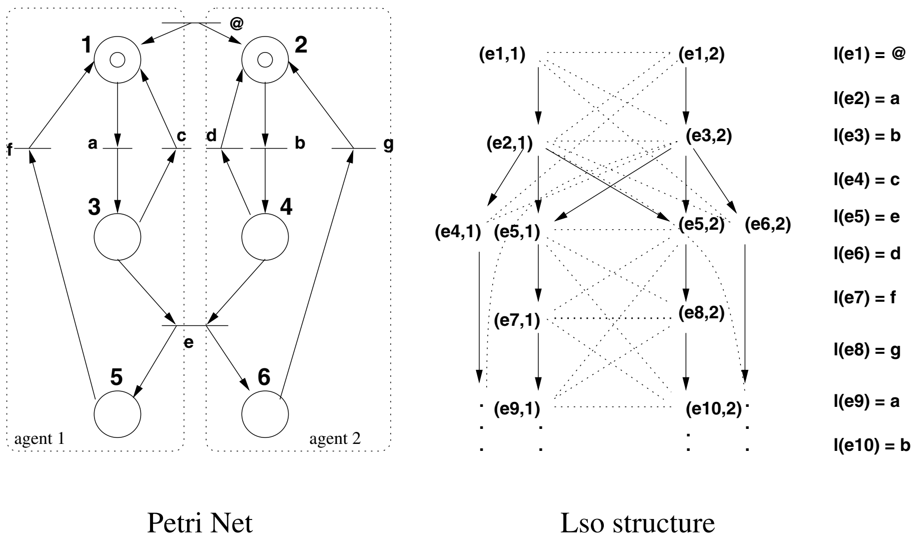


Fig. 2. Petri Net together with the corresponding lso-structure

4.1 Modal Logic for MAS

In this section we introduce the language of modal logic for reasoning about MASs. The language is interpreted over abstract models, based on lso-structures extended with selected runs. Then, we show how to restrict the class of models to these corresponding to our architecture, defined in Section 3. We use temporal operators corresponding to the relations $\rightarrow$ and $\rightarrow^*$. Since each lso identifies uniquely the number of its agent, the modal operators expressing knowledge, intentions, and goals do not need to be indexed with agents's numbers.

Let $PV = \{p_1, p_2, \ldots\} \cup \{i \mid i \in N\} \cup A$ be a countable set of propositional variables including propositions corresponding to the agents' numbers (i.e., identifiers) and the agents' actions. Let $b \in A$ and let $B \subseteq A$ be a set of actions. The logical connectives $\neg$ and $\wedge$, as well as modalities $\Box$ (causally always), $\otimes_B$ (all causally next w.r.t. B), $do(b)$ (next step in the run along b), and do^* (future in the run), and cognitive operators $Know$ (knowledge), Int (intention), and $Goal$ (goal) will be used. The set of temporal and cognitive formulas is built up inductively:

E1. every member of PV is a temporal formula,

E2. if α and β are temporal (cognitive) formulas, then so are $\neg\alpha$ and $\alpha \wedge \beta$,

E3. if α is a temporal (cognitive) formula, then so are $\Box\,\alpha$, $\otimes_B\,\alpha$, $do_b(\alpha)$, and $do^*(\alpha)$,

K4. if α is a temporal formula, then $Know(\alpha)$, $Goal(\alpha)$, and $Int(\alpha)$ are cognitive formulas.

Notice that cognitive modalities cannot be nested. This means for example that reasoning on knowledge about knowledge of other agents is not allowed. The restriction allows to keep the complexity of the model checking algorithm one exponential in the size of the model for the most "complex" variant of agents' knowledge. Notice that although the ontology is common, agents' goals and decision mechanisms are usually different. Frames are based on lso-structures extended with runs and the cognitive functions.

Definition 4 (frame).
Let $\mathcal{S} = (S, A, \rightarrow, \#, l)$ be the lso-structure corresponding to an lpes. A structure $F_R = (\mathcal{S}, R, KNOW, GOAL, INT)$ is a frame*, where*

- *R is a run of S,*
- *$KNOW : S \longrightarrow 2^S$ is a function,*
- *$GOAL : S \longrightarrow 2^{\mathcal{R}}$ is a function,*
- *$INT : S \longrightarrow 2^{\mathcal{R}}$ is a function.*

The intuition behind the cognitive functions is as follows. $KNOW(s)$ gives the lso's, which the agent at s considers as possible for other agents. $GOAL(s)$ gives the runs satisfying the goal of the agent at s. $INT(s)$ gives the runs the agent wants to follow from s.

Definition 5. *A model is a tuple $\mathcal{M}_R = (F_R, V)$, where $F_R$is a frame and $V : S \longrightarrow 2^{PV}$ is a valuation function such that*

- *$i \in V((e, i))$, for each $(e, i) \in S$ and $i \in PV$,*
- *$a \in V((e, i))$, for each $(e, i) \in S$ with $l(e, i) = a$ and $a \in PV$.*

Let $\mathcal{M}_R = (F_R, V)$ be a model, $s = (e, i) \in S$ be a state, and α be a formula. $\mathcal{M}_R, s \models \alpha$ denotes that the formula α is true at the state s in the model $\mathcal{M}_R$ ($\mathcal{M}_R$ is omitted, if it is implicitly understood). $\mathcal{M}_R \models \alpha$ denotes that the formula α is true at all the minimal states s in the model $\mathcal{M}_R$. $\mathcal{M} \models \alpha$ denotes that the formula α holds in all the models $\mathcal{M}_R$ for $R \in \mathcal{R}$. The notion of $s \models \alpha$ for $s = (e, i) \in S$ is defined inductively:

E1. $s \models p$ iff $p \in V(s)$, for $p \in PV$,
E2. $s \models \neg\alpha$ iff not $s \models \alpha$,
$s \models \alpha \wedge \beta$ iff $s \models \alpha$ and $s \models \beta$,
E3. $s \models \Box\, \alpha$ iff $(\forall s' \in S)$ $(s \rightarrow^* s'$ implies $s' \models \alpha)$,
$s \models \otimes_B\, \alpha$ iff $(\forall s' \in S)$ $(s \rightarrow s'$ and $V(s') \cap A \in B$ implies $s' \models \alpha)$,
$s \models do_b(\alpha)$ iff $s \in R$ implies $(\exists s' \in R)$ $(s \rightarrow s'$, $V(s') \cap A = \{b\}$ and $s' \models \alpha)$,
$s \models do^*(\alpha)$ iff $s \in R$ implies $(\exists s' \in R)$ $(s \rightarrow^* s'$ and $s' \models \alpha)$.
K. $s \models Know(\alpha)$ iff $(\forall s' \in KNOW(s))$ $s' \models \alpha$.
G. $s \models Goal(\alpha)$ iff for each $R' \in GOAL(s)$ there is $s' = (e', i) \in R'$ such that $s \rightarrow^* s'$ and $s' \models \alpha$,
I. $s \models Int(\alpha)$ iff for each $R' \in INT(s)$ there is $s' = (e', i) \in R'$ such that $s \rightarrow^* s'$ and $s' \models \alpha$.

Notice that $s \models Goal(\alpha)$ iff for all $R' \in GOAL(s)$ there is an lso $s' \in R$ of the same agent as lso s such that s' is causally later than s and α holds at s'. The meaning of $s \models Int(\alpha)$ is similar w.r.t. $INT(s)$.

4.2 Specifying Correctness of MAS

In this section we show how to restrict the class of models to these corresponding to the specific case of our architecture and how to specify that MAS behaves correctly.

First, for each model $\mathcal{M}_R$ we assume that $R \in INT(s)$ for each lso $s \in S$. The requirement means that the runs representing intentions of each agent contain the run of "reality". This guarantees that the agents perform on their intentions and the reality runs R are consistent with the intentions of all the agents.

Second, we assume that a finite representation of our models $\mathcal{M}_R$ of MAS is given by a deterministic asynchronous automaton [14], denoted $\mathcal{A}$ and extended with a valuation function V and cognitive functions $KNOW$, INT and $GOAL$ defined below. Automaton $\mathcal{A}$ defines the corresponding event structure (the construction can be found in [21,23]) and consequently the corresponding lso-structure. We consider the most complex case of the agents' knowledge, i.e., the **most recent causal knowledge**, where the agents exchange all the information while executing joint actions and have the full capabilities of updating the knowledge. The construction of the cognitive functions $KNOW$, $GOAL$, INT for agent i is determined by the common ontology Λ (for the most recent causal knowledge case), agent i's goal g_i, and the decision mechanism Dec_i, respectively. The definition of $KNOW$ is as follows: $KNOW((e,i)) = \{(e',j) \in S \mid e'$ is the maximal j-event in the causal past of e, for $j \in N\}$. As to representing $GOAL$, for each $i \in N$ we are given a temporal formula φ_i, which specifies the goal states of agent i. $GOAL((e,i))$ codes the goal of agent i in the following way. Each $R' \in GOAL((e,i))$ contains an occurrence of a local state from some situation of agent i's goal g_i ($g_i \models \varphi_i$), i.e., there is $(e',i) \in R'$ with $(e,i) \to (e',i)$ and the local state corresponding to (e',i) belongs to some situation of g_i. Next, to give a formal definition of INT, we have to assume that for each routine $(z, f, z', prob, cost)$ of the decision mechanism, protocol f is given by a partial function $f : L \rightharpoonup 2^A$ from agent's local states to a subset of its actions with the intended meaning: at state $l \in L$ an action from the set $f(l)$ is executed. $INT((e,i))$ codes the decision mechanism of agent i at lso (e,i) in the following way. Each $R' \in INT((e,i))$ is consistent (w.r.t. the executed actions) with the routine $Dec_i(g_i, c)$, where g_i is agent i's goal and $c = Z(l)$ such that (e,i) is an occurrence of the local state l.

For example, we can specify that if the agents' goals φ_i are consistent with their intentions, then the agents reach their goals:

$$COR(\varphi_1, \ldots, \varphi_N) = (\bigwedge_{i \in N} (i \wedge Goal(\varphi_i) \wedge Int(\varphi_i))) \Rightarrow (\bigwedge_{i \in N} do^*(\varphi_i)).$$

5 Model Checking

In this section we outline how to verify automatically that MAS given by $\mathcal{M}$ (i.e., the class of models $\mathcal{M}_R$), finitely represented by $\mathcal{A}$ extended with V, $KNOW$, INT and $GOAL$, behaves correctly, i.e., $\mathcal{M} \models COR(\varphi_1, \ldots, \varphi_N)$.

First, we give an intuitive description of the construction. Our aim is to define a finite quotient structure of $\mathcal{M}$, which preserves the formulas of our language. It has been shown in [14] that it is possible to define constructively a deterministic asynchronous automaton (called gossip), which keeps track about the latest information the agents have about each other. Therefore, the quotient structure is obtained as the global state space of the automaton $\mathcal{B}$ being a product of $\mathcal{A}$ and the gossip automaton $\mathcal{G}$. This structure preserves **all** the temporal and un-nested cognitive formulas of our language. Then, for each routine $r = (z, f, z', prob, cost)$, given by Dec_i, we mark green all the transitions in $\mathcal{B}$, which are consistent with the actions of the protocol f. This is possible because the protocols use as premises local states, which are described by formulas preserved by $\mathcal{B}$. Next, we check whether each concurrency-fair sequence (defined below) of states and green transitions (representing a run) in $\mathcal{G}$ intersects a state, which $i-$local component satisfies φ_i for each $i \in N$. If so, then each run consistent with the intentions and goals is successful, which proves $COR(\varphi_1, \ldots, \varphi_N)$. The detailed construction follows. We abuse the symbol N using it for both the natural number or the set $\{1, ..., N\}$, which is always clear from the context.

Definition 6. *An asynchronous automaton (AA) over a distributed alphabet* $(A_1, \ldots, A_N)$ *is a tuple* $\mathcal{A} = (\{S_i\}_{i\in N}, \{\xrightarrow{a}\}_{a\in A}, S_0, \{S_i^F\}_{i\in N})$, *where*

- S_i *is a set of local states of process* i,
- $\xrightarrow{a} \subseteq S_{agent(a)} \times S_{agent(a)}$, *where* $S_{agent(a)} = \Pi_{i\in agent(a)} S_i$,
- $S_0 \subseteq G_{\mathcal{A}} = \Pi_{i\in N} S_i$ *is the set of initial states,*
- $S_i^F \subseteq S_i$ *is the set of final states of process* i, *for each* $i \in N$.

We deal with deterministic AA's extended with valuation functions $V : G_{\mathcal{A}} \to 2^{PV}$, and functions $GOAL$, INT and $KNOW$ defined on the lso structure corresponding to $\mathcal{A}$. (Alternatively, we could have defined $V_i : S_i \to 2^{PV_i}$ and require that the propositions PV are divided into N disjoint subsets PV_i local to the agents.)

For a global state $g \in G_{\mathcal{A}}$ and $K \subseteq N$ by $g\mid_K$ we mean the projection of g to the local states of processes in K. Let $\Rightarrow_{\mathcal{A}} \subseteq G_{\mathcal{A}} \times A \times G_{\mathcal{A}}$ be the transition relation in the global state space $G_{\mathcal{A}}$ defined as follows: $g \overset{a}{\Rightarrow}_{\mathcal{A}} g'$ iff $(g\mid_{agent(a)}, g'\mid_{agent(a)}) \in \xrightarrow{a}$ and $g\mid_{N\setminus agent(a)} = g'\mid_{N\setminus agent(a)}$.

An *execution sequence* $w = a_0 \ldots a_n \in A^*$ of $\mathcal{A}$ is a finite sequence of actions s.t. there is a sequence of global states and actions $\xi = g_0\, a_0\, g_1\, a_1\, g_2\, \ldots a_{n-1} g_n$ of $\mathcal{A}$ with $g_0 \in S_0$, $g_n \in \Pi_{i\in N} S_i^F$, and $g_i \overset{a_i}{\Rightarrow}_{\mathcal{A}} g_{i+1}$, for each $i < n$. A word w is said to be *accepted* by $\mathcal{A}$ if w is an execution sequence of $\mathcal{A}$. A sequence ξ is *concurrency-fair (cf)* if it is maximal and there is no action, which is eventually continuously enabled and concurrent with the executed actions in ξ.

In order to define the lso-structure semantics of automaton $\mathcal{A}$, we first define the configuration structure $CS = (C_{\mathcal{A}}, \to)$ corresponding to $\mathcal{A}$. Then, the lpes and the lso-structure is induced by CS. Since $\mathcal{A}$ is deterministic, the configurations of CS can be represented by Mazurkiewicz traces [12].

5.1 Trace Semantics of AA's

By an *independence alphabet* we mean any ordered pair (A, I), where $I \subseteq A^2 \setminus D$ (D was introduced in the former Section). Define $\equiv$ as the least congruence in the (standard) string monoid $(A^*, \circ, \epsilon)$ such that $(a, b) \in I \Rightarrow ab \equiv ba$, for all $a, b \in \Sigma$ i.e., $w \equiv w'$, if there is a finite sequence of strings $w_1, \ldots, w_n$ such that $w_1 = w$, $w_n = w'$, and for each $i < n$, $w_i = uabv$, $w_{i+1} = ubav$, for some $(a, b) \in I$ and $u, v \in A^*$. Equivalence classes of $\equiv$ are called *traces* over (A, I). The trace generated by a string w is denoted by $[w]$. We use the following notation: $[A^*] = \{[w] \mid w \in A^*\}$. Concatenation of traces $[w], [v]$, denoted $[w][v]$, is defined as $[wv]$. The *successor* relation $\rightarrow$ in $[A^*]$ is defined as follows: $[w_1] \rightarrow [w_2]$ iff there is $a \in A$ such that $[w_1][a] = [w_2]$.

Definition 7. *The structure $CS = (C_{\mathcal{A}}, \rightarrow)$ is a configuration structure of the automaton $\mathcal{A}$, where*

- *$[w] \in C_{\mathcal{A}}$ iff w is an execution sequence of $\mathcal{A}$,*
- *$\rightarrow$ is the trace successor relation in $C_{\mathcal{A}}$.*

Configuration $[w]$ is i-local iff for each $w' \in A^*$ with $[w] = [w']$ there is $a \in A_i$ and $w'' \in A^*$ such that $w' = w''a$.

The definition of the lpes and the lso-structure corresponding to $\mathcal{A}$ can be obtained from CS. When we represent configurations by traces, the same configurations can belong to different AA's. Therefore, we adopt the convention that $M_{\mathcal{A}}(c)$ denotes the global state in automaton $\mathcal{A}$ corresponding to the configuration c. Let $Max([w]) = \{a \mid [w] = [w'a]$ for some $w' \in A^*\}$, $\downarrow^i [w]$ be the maximal i-local configuration $[w']$ such that $[w'] \rightarrow^* [w]$, and $agent([w]) = \{i \in N \mid [w]$ is i-local$\}$. Moreover, by saying that $c \equiv_l c'$ in $\mathcal{A}$ we mean that $f_l^{\mathcal{A}}(c) = f_l^{\mathcal{A}}(c')$, where

- $f_g^{\mathcal{A}} : C_{\mathcal{A}} \longrightarrow G_{\mathcal{A}} \times 2^A$ such that $f_g^{\mathcal{A}}(c) = (M_{\mathcal{A}}(c), Max(c))$,
- $f_l^{\mathcal{A}} : C_{\mathcal{A}} \longrightarrow \Pi_{i=0}^{N}(G_{\mathcal{A}} \times 2^A)$ with $f_l^{\mathcal{A}}(c) = (f_g^{\mathcal{A}}(c), f_g^{\mathcal{A}}(\downarrow^1 (c)), \ldots, f_g^{\mathcal{A}}(\downarrow^N (c)))$.

It has been shown in [23] that the equivalence $\equiv_l$ preserves the temporal and (non-nested) knowledge formulas of our language.

5.2 Gossip Automaton

Let $\mathcal{A}$ be an AA. For each $i, j \in N$ define the functions:

- $latest_{i \rightarrow j} : C_{\mathcal{A}} \longrightarrow \bigcup_{a \in A}\{\xrightarrow{a}\}$ is defined as follows: $latest_{i \rightarrow j}(c) = (S, S')$ iff (S, S') is the latest transition executed by agent j in $\downarrow^i c$, i.e., if $\downarrow^j\downarrow^i c = \downarrow e$, then event e corresponds to the transition (S, S').
- $latest_i : C_{\mathcal{A}} \longrightarrow 2^N$ is defined as follows: $latest_i(c) = K$ iff $\downarrow^i\downarrow^l (c) \subseteq \downarrow^i\downarrow^k (c)$, for each $l \in N$ and $k \in K$.

Intuitively, $latest_{i \rightarrow j}(c)$ gives the most recent transition in which agent j participated in the i-history of c, i.e., in $\downarrow^i c$, whereas $latest_i(c)$ gives the set of agents, which have the most recent information about agent i.

Theorem 8 ([14]). *There exists a deterministic asynchronous automaton, called Gossip automaton, $\mathcal{G} = (\{T_i\}_{i \in N}, \{\xrightarrow{a}\}_{a \in A}, T_0, \{T_i^F\}_{i \in N})$ such that:*

- *$T_i^F = T_i$, for all $i \in N$,*
- *$\mathcal{G}$ accepts all the words of A^*,*
- *There are effectively computable functions:*
  $gossip : T_1 \times \ldots \times T_N \times N \times N \longrightarrow \bigcup_{a \in A}\{\xrightarrow{a}\}$ such that for each $c \in [A^*]$ and every $i, j \in N$, $latest_{i \to j}(c) = gossip(t_1, \ldots, t_N, i, j)$, where $M_{\mathcal{G}}(c) = (t_1, \ldots, t_N)$.
  $gossip1 : T_1 \times \ldots \times T_N \times N \longrightarrow 2^N$ such that for each $c \in [A^*]$ and every $i \in N$, $latest_i(c) = gossip1(t_1, \ldots, t_N, i)$, where $M_{\mathcal{G}}(c) = (t_1, \ldots, t_N)$.

Each agent in the gossip automaton has $2^{O(N^2 logN)}$ local states. Moreover, the functions *gossip* and *gossip*1 can be computed in time, which is polynomial in the size of N.

Consider the asynchronous automaton $\mathcal{B}$, which is the product of automaton $\mathcal{A}$ and automaton $\mathcal{G}$. We assume that all the local states of $\mathcal{A}$ are final. This is also the case for $\mathcal{G}$. Then, each state of the global state space of $\mathcal{B}$ is of the following form $(l_1, \ldots, l_N, t_1, \ldots, t_N)$, where $l_i \in S_i$ and $t_i \in T_i$, for $i \in N$. The transition relation $\Rightarrow_{\mathcal{B}}$ is defined as follows: $(l_1, \ldots, l_N, t_1, \ldots, t_N) \overset{a}{\Rightarrow}_{\mathcal{B}} (l'_1, \ldots, l'_N, t'_1, \ldots, t'_N)$ iff $(l_1, \ldots, l_N) \overset{a}{\Rightarrow}_{\mathcal{A}} (l'_1, \ldots, l'_N)$ and $(t_1, \ldots, t_N) \overset{a}{\Rightarrow}_{\mathcal{G}} (t'_1, \ldots, t'_N)$.

Notice that automaton $\mathcal{B}$ accepts exactly all the words accepted by $\mathcal{A}$.

Theorem 9. *Let $c, c' \in C_{\mathcal{A}}$. If $M_{\mathcal{B}}(c) = M_{\mathcal{B}}(c')$, then $c \equiv_l c'$ in $\mathcal{A}$.*

Proof. Let $M_{\mathcal{B}}(c) = M_{\mathcal{B}}(c') = (l_1, \ldots, l_N, t_1, \ldots, t_n)$. Obviously, $M_{\mathcal{A}}(c) = M_{\mathcal{A}}(c') = (l_1, \ldots, l_N)$. Notice that $i \in agent(c)$ iff $i \in agent(c')$ iff $i \in gossip1(t_1, \ldots, t_N, j)$ for each $j \in N$. Therefore, $a \in Max(c)$ iff $a \in Max(c')$ iff $gossip(t_1, \ldots, t_N, i, i) \in \xrightarrow{a}$ for $i \in agent(a) \cap agent(c)$. $M_{\mathcal{A}}(\downarrow^i c) = M_{\mathcal{A}}(\downarrow^i c') = (s_1, \ldots, s_N)$ iff $gossip(t_1, \ldots, t_N, i, j) = (S, S')$, where $S'|_j = s_j$ for all $j \in N$. Finally, $a \in Max(\downarrow^i c)$ iff $a \in Max(\downarrow^i c')$ iff $gossip(t_1, \ldots, t_N, j, i) \in \xrightarrow{a}$ for some $j \in agent(c)$.

Therefore, model checking can be performed over the structure $F_{\mathcal{M}} = (W, \to, V)$, where $W = \{(M_{\mathcal{B}}(c), i) \mid i \in agent(c)$ for c-local, and $i = *$, otherwise$\}$, $(M_{\mathcal{B}}(c), i) \xrightarrow{a} (M_{\mathcal{B}}(c'), j)$ iff $M_{\mathcal{B}}(c) \overset{a}{\Rightarrow}_{\mathcal{B}} M_{\mathcal{B}}(c')$, and $p_i \in V(M_{\mathcal{B}}(c), i)$ iff $p_i \in V(M_{\mathcal{A}}(c))$.

5.3 Model Checking over $F_{\mathcal{M}}$

Model checking is performed according to the following rules:

- Formulas are assigned first to the states $W' \subseteq W$ corresponding to the local configurations, i.e., of the form $(M_{\mathcal{B}}(c), i)$, where $i \neq *$. Then, the local components of the states corresponding to the global configurations are labelled with the corresponding (inherited from local configurations) formulas.
  Let $c \in C_{\mathcal{B}}$ and $f_l^{\mathcal{B}}(c) = ((M_g, A_g), (M_1, A_1), \ldots, (M_N, A_N))$.

– $(M_{\mathcal{B}}(c), i) \models \bigcirc_B \alpha$ iff there is $b \in B$ with $i \in agent(b)$ and there is a finite path $g_0 b_0 g_1 b_1 \ldots b_{n-1} g_n$ in W such that $g_0 = (M_{\mathcal{B}}(c), i)$, $g_n \in W'$, and $b_{n-1} = b$, $i \notin agent(b_j)$ for $j < n-1$, and $g_n \models \alpha$.
– $(M_{\mathcal{B}}(c), i) \models \otimes_B \alpha$ iff $(M_{\mathcal{B}}(c), i) \models \neg \bigcirc_B \neg\alpha$.
– $(M_{\mathcal{B}}(c), i) \models \Box\alpha$ iff $(M_{\mathcal{B}}(c), i) \models \alpha$ and $(M_{\mathcal{B}}(c), i) \models \otimes_A^j \alpha$, for each $0 < j \leq |W'|$,
– $(M_{\mathcal{B}}(c), i) \models Know(\alpha)$ iff $(M_{\mathcal{B}}(c'), k) \models \alpha$ for all $k \in N$ and k-local $c' \in C_{\mathcal{B}}$ with $f_g^{\mathcal{B}}(c') = (M_k, A_k)$.
– For each $(M_{\mathcal{B}}(c), i)$ with $i \neq *$ the function INT allows to compute the set of actions, which labeles the transitions executed next in the runs of of $INT((c, i))$. The local transitions (i.e., belonging to one agent only) labelled by these actions are marked green. The joint transitions to be marked green must be consistent with the functions INT at all the starting local configurations of c. Formally, each transition $(M_{\mathcal{B}}(c), i) \xrightarrow{a} (M_{\mathcal{B}}(c'), j)$ is labelled green iff for each $k \in agent(a)$ and all runs $R' \in INT((\downarrow^k c), k)$ a is executed next in R', i.e., $do_a(true)$ holds.
– For each concurrency-fair sequence of states and green transitions, which is consistent with some $Int(\varphi_i)$, i.e., contains a state with $i-$local component labelled φ_i, we check whether for each formula $Goal(\varphi_i)$ there is a state with the $i-$local component labelled φ_i.
Intuitively, this means that we check whether each complete computation consistent with the intentions of all the agents and satisfying goal of one of them, satisfies the goals of all of them.

Theorem 10. *The model checking algorithm for formula* $\psi \overset{def}{=} COR(\varphi_1, \ldots, \varphi_N)$ *over automaton* $\mathcal{A}$ *of* N*-agents is of the complexity* $((|\psi| - m) + m \times |A|) \times |G_{\mathcal{A}}|^N \times 2^{O(N^3 logN)}$*, where* $|G_{\mathcal{A}}|$ *is the size of the global state space of* $\mathcal{A}$ *and* m *is the number of the subformulas of* $\{\varphi_1, \ldots, \varphi_N\}$ *of the form* $\otimes \phi$.

Proof. (Sketch) The complexity follows from the upper bound on the size of the gossip automaton, given in [14], and the complexity of checking the formulas of our language over the finite quotient structure. It is assumed that the complexity of each Dec_i (inducing INT) is in the order of the global state space of automaton $\mathcal{B}$.

Notice that due the partial order semantics, partial order reductions are applicable to $G_{\mathcal{A}}$ (see [21]).

References

1. S. Ambroszkiewicz. On the concepts of rationalizability in games. To appear in Annals of Operations Research 2000.
2. S. Ambroszkiewicz, O. Matyja, and W. Penczek. "Team Formation by Self-Interested Mobile Agents." In Proc. 4-th Australian DAI-Workshop, Brisbane, Australia, Springer LNAI 1544, 1998.

3. DARPA Agent Markup Language (DAML) http://www.darpa.mil/iso/ABC/BAA0007PIP.htm
4. The Foundation for Intelligent Physical Agents (FIPA) http://drogo.cselt.it/fipa/
5. GMA FOCUS and IKV++, http://www.fokus.gmd.de/research/cc/ima/climate.
6. GRASSHOPPER http://www.ikv.de/products/grasshopper
7. T. R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In Formal Ontology Analysis and Knowledge Representation, (N. Guarino and R. Poli Eds.) Kluwer Academic Publishers 1994.
8. N. Guarino. The Ontological level. In R. Casi, B. Smith and G. White (Eds.), *Philosophy and the Cognitive Science.* Hölder-Pichler-Tempsky, Vienna, 1994.
9. R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. Reasoning about knowledge, MIT Press, 1995.
10. K. Lodaya, R. Ramanujam, P.S. Thiagarajan, "Temporal logic for communicating sequential agents: I", Int. J. Found. Comp. Sci., vol. 3(2), 1992, pp. 117–159.
11. T. Magedanz, M. Breugst, I. Busse, S. Covaci. Integrating Mobile Agent Technology and CORBA Middleware. AgentLink Newsletter 1, Nov. 1998, http://www.agentlink.org
12. A. Mazurkiewicz, Basic notions of trace theory, LNCS 354, pp. 285–363, 1988.
13. OMG MASIF, Mobile Agent Systems Interoperability Facility, ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf
14. M. Mukund, and M. Sohoni. Keeping track of the latest gossip: Bounded timestamps suffice, *FST&TCS'93, LNCS* **761**, 1993, 388-199.
15. Pegaz - http://www.ipipan.waw.pl/mas/pegaz/.
16. M. Huhn, P. Niebert, and F. Wallner, "Verification based on local states", LNCS 1384, pp. 36–51, 1998.
17. The Ontology Interchange Language: http://www.ontoknowledge.org/oil/.
18. Ontobroker: http://ontobroker.aifb.uni-karlsruhe.de/
19. W. Penczek, A temporal logic for event structures, *Fundamenta Informaticae* XI, pp. 297–326, 1988.
20. W. Penczek, A Temporal Logic for the Local Specification of Concurrent Systems. *Information Processing IFIP-89*, pp. 857–862, 1989.
21. W. Penczek. Model checking for a Subclass of Event Structures, Proc. of TACAS'97, LNCS 1217, Springer-Verlag, pp. 145–164, 1997.
22. W. Penczek. A temporal approach to causal knowledge, Logic Journal of the IGPL, Vol. 8, Issue 1, pp. 87–99, 2000.
23. W. Penczek and S. Ambroszkiewicz, Model checking of local knowledge formulas, Proc. of FCT'99 Workshop on Distributed Systems, Vol. 28 in Electronic Notes in Theoretical Computer Science, 1999.
24. A. Rao, M. Singh, and M. Georgeff. Formal Methods in DAI: Logic-Based Representation and Reasoning. In G. Weiss (Ed.) Multiagent Systems. A modern approach to Distributed Artificial Intelligence.
25. A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI–architecture. In Proc. KR'91, pp. 473–484, Cambridge, Mass., April 1991, Morgan Kaufmann.
26. Winskel, G., An Introduction to Event Structures, LNCS 354, Springer - Verlag, pp. 364–397, 1989.
27. L. Wittgenstein. *Philosophical Investigations.* Basil Blackwell, pp. 20–21, 1958.

Verification within the KARO Agent Theory[⋆]

Ullrich Hustadt[1], Clare Dixon[1], Renate A. Schmidt[2], Michael Fisher[1], John-Jules Meyer[3], and Wiebe van der Hoek[3]

[1] Centre for Agent Research and Development,
Manchester Metropolitan University, UK
{U.Hustadt,C.Dixon,M.Fisher}@doc.mmu.ac.uk

[2] Department of Computer Science,
University of Manchester, UK
schmidt@cs.man.ac.uk

[3] Department of Computer Science,
University of Utrecht and Amsterdam, The Netherlands
{jj,wiebe}@cs.uu.nl

Abstract. This paper discusses automated reasoning in the KARO framework. The KARO framework accommodates a range of expressive modal logics for describing the behaviour of intelligent agents. We concentrate on a core logic within this framework, in particular, we describe two new methods for providing proof methods for this core logic, discuss some of the problems we have encountered in their design, and present an extended example of the use of the KARO framework and the two proof methods.

1 Introduction

The spread of computer technology has meant that more dynamic, complex and distributed computational systems are being developed. Understanding and managing such complexity is notoriously difficult and it is to support this that the agent-based systems paradigm was introduced. In order to reason about agent-based systems, a number of theories of rational agency have been developed, for example the BDI [20] and KARO [23] frameworks.

The KARO logic (for Knowledge, Abilities, Results and Opportunities) falls within a tradition of the use of *modal logic(s)* for describing the behaviour of intelligent agents. As such it builds upon an even longer tradition of philosophical logic where modal logic is employed to describe intensional notions such as knowledge, belief, time, obligation, desire, etc. These very notions are deemed to be appropriate for specifying, designing and implementing intelligent agents [25]. When following first-order logic approaches to the specification of these intensional concepts (such as the situation calculus) one has to build in the relevant properties of these concepts in an ad hoc manner, whereas in modal logic

[⋆] This research was supported by a travel grant of the Netherlands Organization for Scientific Research (NWO) and the British Council under the UK-Dutch Joint Scientific Research Programme JRP 442.

this can be done systematically using modal correspondence theory. Moreover, the availability of modal operators yields expressions in modal logic that are more concise than their first-order counterparts. Besides this natural adequacy of modal logic in this context, also computational issues are important: generally, propositional modal logics may be viewed as decidable fragments of full first-order logic, and the computational properties of (uncombined) modal logics are well-investigated [6,11].

A popular approach to formal verification, particularly of temporal properties, concerns model-checking [4]. This is appropriate when a (finite-state) structure exists (or can be generated efficiently) upon which logical formulae can be tested. In particular, model checking verifies that the structure is a model for the required formula. In our case, we are interested in verifying properties of logical specifications, rather than finite-state structures. Although such models can be built directly from formulae, this is usually expensive. In addition, little work has been carried out concerning model-checking for the types of complex multi-modal logics we require (an exception being the work described in [2]). Consequently, we are here interested in carrying out verification via logical proof.

Thus, the aim of this paper is to examine proof methods for the KARO framework [23]. In particular, we study two approaches to the problem of proof which will be presented in Sections 3 and 4:

– proof methods for the fusion of $\mathcal{PDL}$ and $\mathsf{S5}_{(m)}$ based upon translation to classical logic and first-order resolution; and
– representation of KARO in terms of the fusion of CTL and $\mathsf{S5}_{(m)}$ and proof methods by direct clausal resolution on this combined logic.

Both approaches provide decision procedures for the particular combination of logics under consideration. Thus, unlike approaches which make use of full first-order logic, *unprovability* of a formulae with respect to a agent specification can be shown by each of two approaches automatically without reliance on model-theoretic consideration outside the actual proof method.

We illustrate how the KARO framework and the two approaches can be used by studying a small block world example in Section 5. We conclude with a discussion of the relative strength of the two approaches and a possible reconciliation of the two approaches.

2 Basic KARO Elements

We base our formal methods on the KARO logic [16,22], a formal system that may be used to *specify*, *analyse* and *reason about* the behaviour of rational agents. In this paper we concentrate on one particular variant of the KARO framework and define a core subsystem for which we are able to provide sound, complete, and terminating inference systems.

Formally, the language of the KARO framework is defined over three primitive types: (i) a set of countably infinite atomic propositional variables, (ii) a set

Ag of agent names (a finite subset of the positive integers), and (iii) a set $\mathsf{Ac_{at}}$ of countably infinite atomic action variables.

Formulae are defined inductively as follows.

– $\top$ is an atomic propositional formula;
– $\varphi \vee \psi$ and $\neg\varphi$ are propositional formula provided so are φ and ψ;
– $\mathbf{K}_i\varphi$ (knowledge), $\langle \mathtt{do}_i(\alpha)\rangle\varphi$ (achievement of results by actions), $\mathbf{A}_i\alpha$ (ability), $\mathbf{O}_i\alpha$ (opportunity), $\mathbf{W}^s_i\varphi$ (selected wish), and $\Diamond_i\varphi$ (implementability) are propositional formulae, provided i is an agent name, α is an action formula and φ is a propositional formula;
– $\mathtt{id}$ (skip) is an atomic action formula;
– $\alpha \vee \beta$ (non-deterministic choice), $\alpha\,;\beta$ (sequencing), $\varphi!$ (confirm), $\alpha^{(n)}$ (bounded repetition), and $\alpha^\star$ (unbounded repetition) are action formulae, provided α and β are action formulae, φ is a propositional formula, and n is a natural number.

Implicit connectives include $\bot, \wedge, \rightarrow, \ldots$ for propositional formulae, the duals of $\mathbf{K}_i$ and $\langle \mathtt{do}_i(\alpha)\rangle$ (denoted by $[\mathtt{do}_i(\alpha)]$), as well as $\mathbf{P}_i(\alpha,\varphi) = \langle \mathtt{do}_i(\alpha)\rangle\varphi \wedge \mathbf{A}_i\alpha$.

The semantics of the core KARO logic is based on *interpretations* $\mathcal{M} = (W, V, D, I, M)$, where (i) W is a non-empty set of worlds, (ii) V maps propositional variables to subsets of W, (iii) for every $i \in \mathsf{Ag}$ and every $a \in \mathsf{Ac_{at}}$, D contains a binary relation $r_{(i,a)}$ on W and a subset $c_{(i,a)}$ of W, (iv) I contains an equivalence relation K_i on W for each agent $i \in \mathsf{Ag}$, and (v) M contains a serial relation relation W_i on W for each agent $i \in \mathsf{Ag}$. Following the characterisation of agent theories in the introduction, D, I, and M comprise the dynamic, informational, and motivational components in the semantics of the core KARO logic. The relations r and sets c are extended to $\mathsf{Ag} \times \mathsf{Ac}$-sorted relations r^* and sets c^* in a way standard for dynamic logic.

The semantics of well-formed formulae of the KARO logic is defined as follows.

$$
\begin{array}{ll}
\mathcal{M}, w \models \top & \\
\mathcal{M}, w \models p & \text{iff } w \in V(p) \\
\mathcal{M}, w \models \neg\varphi & \text{iff } \mathcal{M}, w \not\models \varphi \\
\mathcal{M}, w \models \varphi \vee \psi & \text{iff } \mathcal{M}, w \models \varphi \text{ or } \mathcal{M}, w \models \psi \\
\mathcal{M}, w \models [\mathtt{do}_i(\alpha)]\varphi & \text{iff } \forall v \in W\ ((w,v) \in r^*_{(i,\alpha)} \rightarrow \mathcal{M}, v \models \varphi) \\
\mathcal{M}, w \models \mathbf{A}_i\alpha & \text{iff } w \in c^*_{(i,\alpha)} \\
\mathcal{M}, w \models \mathbf{O}_i\alpha & \text{iff } \mathcal{M}, w \models \langle \mathtt{do}_i(\alpha)\rangle\top \\
\mathcal{M}, w \models \mathbf{W}^s_i\varphi & \text{iff } \forall v \in W\ ((w,v) \in W_i \rightarrow \mathcal{M}, v \models \varphi) \\
\mathcal{M}, w \models \mathbf{K}_i\varphi & \text{iff } \forall v \in W\ ((w,v) \in K_i \rightarrow \mathcal{M}, v \models \varphi) \\
\mathcal{M}, w \models \Diamond_i\varphi & \text{iff } \exists k \in \mathbb{N}\ \exists a_1, \ldots, a_k \in \mathsf{Ac_{at}}\ \mathcal{M}, w \models \mathbf{P}_i(a_1; \ldots; a_k, \varphi)
\end{array}
$$

If $\mathcal{M}, w \models \varphi$ we say φ *holds at* w (in $\mathcal{M}$) or φ is *true in* w. A formula φ is *satisfiable* iff there is an interpretation $\mathcal{M}$ and a world w such that $\mathcal{M}, w \models \varphi$.

We refer to the logic defined above as the *KARO logic* even though it does not include all the features of the KARO framework. In this paper we make the following simplifying assumptions: (i) we assume $\mathbf{A}_i\alpha = \langle \mathtt{do}_i(\alpha)\rangle\top$, (ii) we

$$\begin{array}{ll}
\neg\langle \mathtt{do}_i(\alpha)\rangle\psi \Rightarrow [\mathtt{do}_i(\alpha)]\neg\psi & \neg[\mathtt{do}_i(\alpha)]\psi \Rightarrow \langle \mathtt{do}_i(\alpha)\rangle\neg\psi \\
\langle \mathtt{do}_i(\alpha)\rangle(\psi \vee \phi) \Rightarrow \langle \mathtt{do}_i(\alpha)\rangle\psi \vee \langle \mathtt{do}_i(\alpha)\rangle\phi & [\mathtt{do}_i(\alpha)](\psi \wedge \phi) \Rightarrow [\mathtt{do}_i(\alpha)]\psi \wedge [\mathtt{do}_i(\alpha)]\phi \\
\langle \mathtt{do}_i(\alpha \vee \beta)\rangle\psi \Rightarrow \langle \mathtt{do}_i(\alpha)\rangle\psi \vee \langle \mathtt{do}_i(\beta)\rangle\psi & [\mathtt{do}_i(\alpha \vee \beta)]\psi \Rightarrow [\mathtt{do}_i(\alpha)]\psi \wedge [\mathtt{do}_i(\beta)]\psi \\
\langle \mathtt{do}_i(\alpha\,;\beta)\rangle\psi \Rightarrow \langle \mathtt{do}_i(\alpha)\rangle\langle \mathtt{do}_i(\beta)\rangle\psi & [\mathtt{do}_i(\alpha\,;\beta)]\psi \Rightarrow [\mathtt{do}_i(\alpha)][\mathtt{do}_i(\beta)]\psi \\
\langle \mathtt{do}_i(\mathtt{id})\rangle\psi \Rightarrow \psi & [\mathtt{do}_i(\mathtt{id})]\psi \Rightarrow \psi \\
\langle \mathtt{do}_i(\phi!)\rangle\psi \Rightarrow \phi \wedge \psi & [\mathtt{do}_i(\phi!)]\psi \Rightarrow \neg\phi \vee \psi \\
\langle \mathtt{do}_i(\alpha^{(0)})\rangle\psi \Rightarrow \psi & [\mathtt{do}_i(\alpha^{(0)})]\psi \Rightarrow \psi \\
\langle \mathtt{do}_i(\alpha^{(n+1)})\rangle\psi \Rightarrow \langle \mathtt{do}_i(\alpha)\rangle\langle \mathtt{do}_i(\alpha^{(n)})\rangle\psi & [\mathtt{do}_i(\alpha^{(n+1)})]\psi \Rightarrow [\mathtt{do}_i(\alpha)][\mathtt{do}_i(\alpha^{(n)})]\psi
\end{array}$$

Fig. 1. Transformation rules for the core KARO logic

exclude the unbounded repetition operator $\alpha^\star$ and wishes $\mathbf{W}_i^s\varphi$ from the language, and (iii) there is no interaction between the dynamic and informational component. This fragment of the KARO logic is called the *core KARO logic*.

The proof-theoretical requirements induced by the implementability operator are quite strong, and both approaches which will presented in the following two sections will weaken these requirements in different ways.

3 Proof by Translation

The translation approach to modal reasoning is based on the idea that inference in (combinations of) modal logics can be carried out by translating modal formulae into first-order logic and using conventional first-order theorem proving. There are various different translation morphisms for modal logics whose properties vary with regards the extent to which they are able to map modal logics into first-order logic, the decidability of the fragments of first-order logic into which modal formulae are translated, and the computational behaviour of first-order theorem provers on these fragments [6,13,15,21].

In the following we present a decision procedure for the satisfiability problem in the core KARO logic consisting of three components: (i) a normalisation function which reduces complex action formulae to atomic action subformulae, (ii) a particular translation of normalised formulae into a fragment of first-order logic, and (iii) a resolution-based decision procedure for this fragment.

The normalisation function maps any formula φ of the core KARO logic to its normal form $\varphi\!\downarrow$ under the rewrite rules given in Figure 1. It is straightforward to see that the rewrite relation defined by these rules is confluent and terminating. Thus, the normal form $\varphi\!\downarrow$ of φ is logically equivalent to φ, it is unique, and in the absence of the unbounded repetition operator, $\varphi\!\downarrow$ contains no non-atomic action formulae.

Lemma 1. *The core KARO logic excluding the operator $\Diamond_i$ equivalently reduces to the fusion of multi-modal $\mathsf{K}_{(m)}$ and $\mathsf{S5}_m$.*

The particular translation we use has only recently been proposed by de Nivelle [5] and can be seen as a special case of the T-encoding introduced by Ohlbach [18]. It allows for conceptually simple decision procedures for extensions of K4 by ordered resolution without any reliance on loop checking or similar techniques.

Without loss of generality we assume that the modal formulae under consideration are normalised and in negation normal form. We define a translation function π as follows.

$$\begin{aligned}
\pi(\langle \mathtt{do}_i(a)\rangle\varphi, x) &= \exists y\, (\mathtt{do}_i^a(x,y) \wedge \pi(\varphi,y)) & \pi(\top, x) &= \top \\
\pi(\mathbf{K}_i\varphi, x) &= q_{\mathbf{K}_i\varphi}(x) & \pi(p, x) &= q_p(x) \\
\pi(\mathbf{O}_i\alpha, x) &= \pi(\langle \mathtt{do}_i(\alpha)\rangle\top, x) & \pi(\neg\varphi, x) &= \neg\pi(\varphi, x) \\
\pi(\mathbf{A}_i\alpha, x) &= \pi(\langle \mathtt{do}_i(\alpha)\rangle\top, x) & \pi(\varphi \vee \psi, x) &= \pi(\varphi, x) \vee \pi(\psi, x) \\
\pi(\Diamond_i\varphi, x) &= \exists y\, \pi(\varphi, y)
\end{aligned}$$

a is an atomic action, p is a propositional variable, q_p is a unary predicate symbol uniquely associated with p, $q_{\mathbf{K}_i\varphi}$ is a predicate symbol uniquely associated with $\mathbf{K}_i\varphi$, and $\mathtt{do}_i^a$ is a binary predicate symbol associated with a and i which represent the relation $r_{(i,a)}$ in the semantics.

Finally, let $\Pi(\psi)$ be the formula

$$\exists x\, \pi(\psi, x) \wedge \textstyle\bigwedge_{\mathbf{K}_i\varphi \in \Gamma_{\mathbf{K}}(\psi)} \mathrm{Ax}(\mathbf{K}_i\varphi),$$

where $\Gamma_{\mathbf{K}}(\psi)$ is the set of subformulae of the form $\mathbf{K}_i\varphi$ in ψ, and $\mathrm{Ax}(\mathbf{K}_i\varphi)$ is the formula

$$\begin{aligned}
&\forall x\, (q_{\mathbf{K}_i\varphi}(x) \leftrightarrow \forall y\, (K_i(x,y) \rightarrow \pi(\varphi, y))) \\
\wedge\, &\forall x, y\, ((q_{\mathbf{K}_i\varphi}(x) \wedge K_i(x,y)) \rightarrow q_{\mathbf{K}_i\varphi}(y)) \\
\wedge\, &\forall x, y\, ((q_{\mathbf{K}_i\varphi}(y) \wedge K_i(x,y)) \rightarrow q_{\mathbf{K}_i\varphi}(x)) \wedge \forall x\, K_i(x,x).
\end{aligned}$$

Based on the close correspondence between the translation morphism Π and the semantics of the core KARO logic it is possible to prove the following.

Theorem 2. *Let ψ be a formula in the core KARO logic excluding the operator $\Diamond_i$. Then $\Pi(\psi)$ is first-order satisfiable iff there exists a model $\mathcal{M}$ and a world w such that $\mathcal{M}, w \models \psi$.*

Proof. The only problem in this theorem is caused by the fact that $\Pi(\psi)$ does not ensure that the relations K_i in a first-order model of $\Pi(\psi)$ are not necessarily equivalence relations while this is the fact for the corresponding relations K_i in the modal model. This problem can be overcome along the lines of de Nivelle [5] or Hustadt et al. [14].

Using certain structural transformation techniques, described for example in [6, 19], $\Pi(\psi)$ can be embedded into a number of solvable clausal classes, for example, the classes $\mathcal{S}^+$ [9] and DL* [6]. In the following, by $\mathrm{CL}_{\mathrm{DL}^*}(\Pi(\psi))$ we denote an embedding of $\Pi(\psi)$ into the class DL*. A decision procedures for DL* can be formulated in the resolution framework of Bachmair and Ganzinger [1] using an ordering refinement of resolution.

Theorem 3 (Soundness, completeness, and termination [6]). *Let ψ be a formula of the core KARO logic excluding $\Diamond_i$ and let $N = \mathrm{CL}_{\mathrm{DL}^*}(\Pi(\psi))$. Let $\succ$ be any ordering which is compatible with the strict subterm ordering and let $\mathsf{R}^{\succ}$ be the ordered resolution calculus restricted by $\succ$. Then:*

1. *Any derivation from N in $\mathsf{R}^{\succ}$ terminates in double exponential time.*
2. *φ is unsatisfiable iff the empty clause can be derived from N.*

All the results presented also hold for the core KARO logic including $\Diamond_i$ under our assumption that $\mathbf{A}_i\alpha = \langle \mathsf{do}_i(\alpha)\rangle\top$. If we drop this assumption, then the translation by π and Π does no longer preserve satisfiability. Although it is possible to devise alternative translations π' and Π' (into second-order logic) such that Theorem 2 holds for the core KARO logic including $\Diamond_i$, Theorem 3 would be invalid for Π'. This is not difficult to see if we consider the problem of showing that $\Diamond_i\varphi$ is not true at a world w in an interpretation $\mathcal{M}$. This is the case iff $\forall k \in \mathbb{N}\ \forall a_1, \dots, a_k \in \mathsf{Ac}_{\mathsf{at}}\ \mathcal{M}, w \not\models \mathbf{P}_i(a_1; \dots ; a_k, \varphi)$. Due to the (universal) quantification over $\mathbb{N}$, proving that $\Diamond_i\varphi$ is not true at a world w requires *inductive theorem proving*. This is outside the scope of first-order resolution and termination is not guaranteed.

Since our emphasis is on practical inference methods to support the deductive verification of agents, we have opted for a simplified core logic.

4 Proof by Clausal Temporal Resolution

Here, we use the simple observation that the use of $\mathcal{PDL}$ in the KARO framework is very similar to the use of branching-time temporal logic. Thus, we attempt to use a simple CTL branching-time temporal logic to represent the dynamic component of the core KARO logic. Dynamic operators and implementability are replaced by CTL formulae by the following rules: for example,

$$\begin{array}{rll}
\langle \mathsf{do}_i(a)\rangle\varphi & \text{is replaced by} & \mathbf{E}\bigcirc(\mathrm{done}_i(a) \wedge \varphi), \\
[\mathsf{do}_i(a)]\varphi & \text{is replaced by} & \mathbf{A}\bigcirc(\mathrm{done}_i(a) \Rightarrow \varphi) \text{ and} \\
\Diamond_i\varphi & \text{is replaced by} & \mathbf{E}\Diamond\varphi,
\end{array}$$

where $\mathrm{done}_i(a)$ is a propositional variable uniquely associated with agent i and atomic action a. Initially we work in the logic CTL, with multi-modal $\mathsf{S5}$, the semantics of which are given in, for example [12], with no interactions.

Formulae in the fusion of CTL and $\mathsf{S5}_{(n)}$ can rewritten into a normal form, called SNF_{karo}, that separates temporal and modal aspects (as is done in [8]). Formulae in SNF_{karo} are of the general form $\mathbf{A}\Box^* \bigwedge_i T_i$ where $\mathbf{A}\Box^*$ is the universal relation (which can be defined in terms of the operators "everyone knows" and "common knowledge") and each T_i is a *clause* of one of the following forms.

$$\begin{array}{lll}
\mathbf{start} \Rightarrow \bigvee_{k=1}^{n} L_k & & (\textit{initial} \text{ clauses}) \\
\bigwedge_{j=1}^{m} L'_j \Rightarrow \mathbf{A}\bigcirc \bigvee_{k=1}^{n} L_k & \bigwedge_{j=1}^{m} L'_j \Rightarrow \mathbf{E}\bigcirc(\bigvee_{k=1}^{n} L_k)_{\langle \mathsf{c}_i\rangle} & (\textit{step} \text{ clauses}) \\
\bigwedge_{j=1}^{m} L'_j \Rightarrow \mathbf{A}\Diamond L & \bigwedge_{j=1}^{m} L'_j \Rightarrow \mathbf{E}\Diamond L_{\langle \mathsf{c}_i\rangle} & (\textit{sometime} \text{ clauses}) \\
\mathbf{true} \Rightarrow \bigvee_{k=1}^{n} M_k^i & & (\mathbf{K}_i \text{ clauses}) \\
\mathbf{true} \Rightarrow \bigvee_{k=1}^{n} L_k & & (\text{literal clauses})
\end{array}$$

where L'_j, L_k, and L are literals and M^i_k are either literals, or modal literals involving the modal operator $\mathbf{K}_i$. Further, each $\mathbf{K}_i$ clause has at least one disjunct that is a modal literal. $\mathbf{K}_i$ clauses are sometimes known as *knowledge clauses.* Each step and sometime clause that involves the $\mathbf{E}$-operator is labelled by an index of the form $\langle \mathsf{c}_i \rangle$ similar to the use of Skolem constants in first-order logic. This index indicates a particular path and arises from the translation of formulae such as $\mathbf{E}(L\mathcal{U}L')$. During the translation to the normal form such formulae are translated into several $\mathbf{E}$ step clauses and a $\mathbf{E}$ sometime clause (which ensures that L' must actually hold). To indicate that all these clauses refer to the same path they are annotated with an index. The outer '$\mathbf{A}\Box^*$' operator that surrounds the conjunction of clauses is usually omitted. Similarly, for convenience the conjunction is dropped and we consider just the set of clauses T_i. We denote the normalisation function into SNF$_{karo}$ by τ.

In the following we present a resolution-based calculus for SNF$_{karo}$. In contrast to the translation approach described in the previous section, this calculus works directly on SNF$_{karo}$ formulae. The inference rules are divided into initial resolution rules, knowledge resolution rules, step resolution rules, and temporal resolution rules, which will be described in the following. We present sufficient rules to understand the following example, the full set of knowledge rules can be found in [8] and the full set of step and temporal resolution rules are given in [3]. In the following, if L is a literal, then $\sim L$ denotes A if $L = \neg A$ and it denotes $\neg L$, otherwise.

Initial Resolution. A literal clause may be resolved with an initial clause (IRES1) or two initial clauses may be resolved together (IRES2) as follows

$$[\text{IRES1}]\quad \frac{\begin{array}{l}\mathbf{true} \Rightarrow (C \vee L)\\ \mathbf{start} \Rightarrow (D \vee \sim L)\end{array}}{\mathbf{start} \Rightarrow (C \vee D)} \qquad [\text{IRES2}]\quad \frac{\begin{array}{l}\mathbf{start} \Rightarrow (C \vee L)\\ \mathbf{start} \Rightarrow (D \vee \sim L)\end{array}}{\mathbf{start} \Rightarrow (C \vee D)}$$

where C and D are disjunctions of literals.

Knowledge Resolution. During knowledge resolution we apply the following rules which are based on the modal resolution system introduced by Mints [17]. In general we may only apply a (knowledge) resolution rule between two literal clauses, a knowledge and a literal clause, or between two knowledge clauses relating to the same modal operator e.g. two $\mathbf{K}_1$ clauses.

$$[\text{KRES1}]\quad \frac{\begin{array}{l}\mathbf{true} \Rightarrow C \vee M\\ \mathbf{true} \Rightarrow D \vee \sim M\end{array}}{\mathbf{true} \Rightarrow C \vee D} \qquad [\text{KRES4}]\quad \frac{\begin{array}{l}\mathbf{true} \Rightarrow C \vee \neg\mathbf{K}_i L\\ \mathbf{true} \Rightarrow D \vee L\end{array}}{\mathbf{true} \Rightarrow C \vee \mathrm{mod}(D)}$$

The function mod(D) used in KRES4 is defined on disjunctions D of literals or modal literals, as follows.

$$\begin{array}{rlrl}\mathrm{mod}(\mathbf{K}_i L) &= \mathbf{K}_i L & \mathrm{mod}(\neg\mathbf{K}_i L) &= \neg\mathbf{K}_i L\\ \mathrm{mod}(A \vee B) &= \mathrm{mod}(A) \vee \mathrm{mod}(B) & \mathrm{mod}(L) &= \neg\mathbf{K}_i{\sim}L\end{array}$$

Two further rules KRES2 and KRES3 allow resolution between $\mathbf{K}_i L$ and $\mathbf{K}_i {\sim} L$, and between $\mathbf{K}_i L$ and ${\sim}L$.

Finally given a clause involving a disjunction of literals or modal literals of the form $\mathbf{K}_i L$ we can remove the $\mathbf{K}_i$ operators (KRES5) obtaining a literal clause for use during step and temporal resolution. For the complete set of modal resolution rules see [8].

Step Resolution. 'Step' resolution consists of the application of standard classical resolution to formulae representing constraints at a particular moment in time, together with simplification rules for transferring contradictions within states to constraints on previous states. Simplification and subsumption rules are also applied. In the following $\mathbf{P}$ is either path operator.

$$[\text{SRES2}]\ \frac{\begin{array}{c} P \Rightarrow \mathbf{E}\bigcirc(F \vee L)_{\langle \mathsf{c}_i \rangle} \\ Q \Rightarrow \mathbf{A}\bigcirc(G \vee {\sim}L) \end{array}}{(P \wedge Q) \Rightarrow \mathbf{E}\bigcirc(F \vee G)_{\langle \mathsf{c}_i \rangle}} \qquad [\text{SRES4}]\ \frac{Q \Rightarrow \mathbf{P}\bigcirc\mathbf{false}}{\mathbf{true} \Rightarrow {\sim}Q}$$

We also allow resolution between two $\mathbf{A}$ step clauses (SRES1) and two $\mathbf{E}$ step clauses (SRES3) with the same index. A step clause may be resolved with a literal clause (where G is a disjunction of literals) and any index is carried to the resolvent to give the following resolution rules.

$$[\text{SRES5}]\ \frac{\begin{array}{r} P \Rightarrow \mathbf{A}\bigcirc(F \vee L) \\ \mathbf{true} \Rightarrow (G \vee {\sim}L) \end{array}}{P \Rightarrow \mathbf{A}\bigcirc(F \vee G)} \qquad \frac{\begin{array}{r} P \Rightarrow \mathbf{E}\bigcirc(F \vee L)_{\langle \mathsf{c}_i \rangle} \\ \mathbf{true} \Rightarrow (G \vee {\sim}L) \end{array}}{P \Rightarrow \mathbf{E}\bigcirc(F \vee G)_{\langle \mathsf{c}_i \rangle}}$$

The complete set of step rules can be found in [3]

Temporal Resolution. During temporal resolution the aim is to resolve one of the sometime clauses, $Q \Rightarrow \mathbf{P}\Diamond L$, with a set of clauses that together imply $\Box{\sim}L$ along the same path, for example a set of clauses that together have the effect of $F \Rightarrow \bigcirc\Box{\sim}L$. However the interaction between the '$\bigcirc$' and '$\Box$' operators makes the definition of such a rule non-trivial and further the translation to SNF_{karo} will have removed all but the outer level of $\Box$-operators. So, resolution will be between a sometime clause and a *set* of clauses that together imply an $\Box$-formula that occurs on the same path, which will contradict the $\Diamond$-clause. The details of these rules can be found in [3].

Theorem 4 (Soundness, completeness, and termination). *Let φ be a formula of the core KARO logic excluding the operator $\Diamond_i$ and let $N = \tau(\varphi)$. Then:*

1. *Any derivation from N terminates.*
2. *φ is unsatisfiable iff N has a refutation by the temporal resolution procedure described above.*

The proofs are analogous to those in [7,8,10].

Again we have excluded the operator $\Diamond_i$ in Theorem 4. The transformation τ which replaces $\Diamond_i\varphi$ by $\mathbf{E}\Diamond\varphi$ does not preserve satisfiability. We are currently investigating an alternative transformation τ' which expands $\Diamond_i\varphi$ with

$\varphi \vee \mathbf{E}(\bigvee_{a \in \mathsf{Ac_{at}}}(c_i^a \wedge \bigcirc done_i^a)))\mathcal{U}\varphi)$ which seems to reflect the semantics of $\Diamond_i\varphi$ more faithfully if we drop the assumption that $\mathbf{A}_i\alpha = \langle \mathtt{do}_i(\alpha)\rangle\top$.

Recall that we have observed in the previous section that the core KARO logic excluding $\Diamond_i$ reduces to the fusion of multi-modal $\mathsf{K}_{(m)}$ and $\mathsf{S5}_m$, while in this section we have used a reduction to the fusion of CTL and $\mathsf{S5}_m$. This may seem unreasonable, since the satisfiability problem in $\mathsf{K}_{(m)}$ is (only) PSPACE-complete, while the satisfiability problem in CTL is EXPTIME-complete. However, as τ is a polynomial reduction mapping, the complexity of testing the satisfiability of $\tau(\varphi)$ is the same as testing the satisfiability of φ. Moreover, the apparent possibility to provide a satisfiability equivalence preserving mapping of $\Diamond_i\varphi$ into SNF_{karo} provides a justification.

5 Eve in a Blocks World

Consider two agents, Adam and Eve, living in a blocks world containing three blocks b, c, and d. We use $\mathrm{on}(X,Y)$ and $\mathrm{clear}(X)$ to describe that a block Y is on top of a block X and that no block is on top of X, respectively. A tower consists of three distinct blocks X_1, X_2, X_3 such that X_3 is clear, X_3 is on X_2, and X_2 is on X_1 (axiom (C_1)). We allow only one atomic action: $\mathrm{put}(X,Y)$, which has the effect of Y being placed on X. Eve has the ability of performing a $\mathrm{put}(X,Y)$ action if and only if X and Y are clear, Y is not identical to X, and Y is not equal to c (axiom (A_1)). The axiom (E_1) describes the effects of performing a put action: After any action $\mathrm{put}(X,Y)$ the block Y is on X and X is no longer clear. The axioms (N_1) to (N_3) describe properties of the blocks world which remain unchanged by performing an action. For example, if block Z is clear and not equal to some block X, then putting some arbitrary block Y (possibly identical to Z) on X leaves Z clear (axiom (N_1)). Additionally, the axioms themselves, except for (I_1), remain true irrespective of the actions which are performed.

$$(A_1) \qquad \mathbf{A}_E\mathrm{put}(X,Y) \equiv (\mathrm{clear}(X) \wedge \mathrm{clear}(Y) \wedge X \neq Y \wedge Y \neq c)$$
$$(E_1) \qquad \rightarrow [\mathtt{do}_i(\mathrm{put}(X,Y))](\mathrm{on}(X,Y) \wedge \neg\mathrm{clear}(X))$$
$$(N_1) \qquad (\mathrm{clear}(Z) \wedge Z \neq X) \rightarrow [\mathtt{do}_i(\mathrm{put}(X,Y))](\mathrm{clear}(Z))$$
$$(N_2) \qquad (\mathrm{on}(V,Z) \wedge Z \neq Y) \rightarrow [\mathtt{do}_i(\mathrm{put}(X,Y)))](\mathrm{on}(V,Z))$$
$$(N_3) \qquad (X = Y) \wedge (U \neq V) \rightarrow [\mathtt{do}_i(\alpha)](X = Y \wedge U \neq V)$$
$$(C_1) \qquad \mathrm{tower}(X_1,X_2,X_3) \equiv \bigwedge_{i \neq j}(X_i \neq X_j) \wedge \mathrm{on}(X_1,X_2) \wedge \mathrm{on}(X_2,X_3) \wedge \mathrm{clear}(X_3)$$
$$(I_1) \qquad \mathbf{K}_E\mathrm{clear}(c) \wedge \mathbf{K}_E\mathrm{clear}(d)$$

In the axioms above i is an element of $\{A, E\}$ where A and E denote Adam and Eve. Recall that in the core KARO logic we identify $\mathbf{A}_i\alpha$ with $\langle\mathtt{do}_i(\alpha)\rangle\top$. Consequently, the axiom (A_1) becomes

$$(A_3') \qquad \langle\mathtt{do}_E(\mathrm{put}(X,Y))\rangle\top \equiv (\mathrm{clear}(X) \wedge \mathrm{clear}(Y) \wedge X \neq Y \wedge Y \neq c)$$

In the following we will prove that the axioms (A_1) to (C_1) together with (I_1) imply that if Eve knows that Adam puts block c on block b, then she knows that she can implement the tower (b, c, d), that is, we show that the assumption

$$(K_1) \qquad \mathbf{K}_E\langle \mathtt{do}_A(\mathrm{put}(b,c))\rangle\top \wedge \neg\mathbf{K}_E\Diamond_E \mathrm{tower}(b,c,d)$$

leads to a contradiction.

Although the problem is presented in a first order setting, as we have a finite domain we can easily form all ground instances of the axioms in our specification. Thus, in the following, an expression ‘on(b, c)’ denotes a propositional variable uniquely associated with the atom on(b, c) in our specification. Due to axiom (N_3) which states that equality and inequality of blocks remains unaffected by Eve’s actions, we can eliminate all equations from the instantiated axioms.

Solving the Eve Example By Translation. We will first show how we obtain a refutation for the specification of Eve’s blocks world using the translation approach. Let ψ be the conjunction of the axioms (A_1) to (C_1), (I_1), and (K_1). Then $\mathrm{CL}_{\mathrm{DL}^*}(\Pi(\psi))$ contains amongst others the following clauses which will be used in our refutation. The axioms from which a particular clause originates are indicated in square brackets to the left of the clause. Recall that $\pi(p, x) = q_p(x)$ where q_p is a unary predicate symbol uniquely associated with the propositional variable p. To simplify our notation we will write ‘on(b, c, x)’ instead of ‘$q_{\mathrm{on}(b,c)}(x)$’. Note that the translation of the axiom (A'_3) and the left conjunction of (K_1) contain existential quantifiers which lead to the introduction of Skolem functions during the transformation to clausal normal form. Consequently, the clauses (1) and (14) contain unary Skolem functions g^d_c and g^c_b, respectively. These Skolem functions are associated with particular actions, namely, put(c, d) and put(b, c), respectively. In addition, the Skolem constant ϵ is introduced by Π itself.

[A'_3]	(1)	$\neg\mathrm{clear}(c,y) \vee \neg\mathrm{clear}(d,y) \vee \mathtt{do}_E^{\mathrm{put}(c,d)}(x, g^d_c(x))_*$
[E_1]	(2)	$\neg\mathtt{do}_A^{\mathrm{put}(b,c)}(x,y)_* \vee \mathrm{on}(b,c,y)$
[E_1]	(3)	$\neg\mathtt{do}_E^{\mathrm{put}(c,d)}(x,y)_* \vee \mathrm{on}(c,d,y)$
[N_1]	(4)	$\neg\mathrm{clear}(c,x) \vee \neg\mathtt{do}_A^{\mathrm{put}(b,c)}(x,y)_* \vee \mathrm{clear}(c,y)$
[N_1]	(5)	$\neg\mathrm{clear}(d,x) \vee \neg\mathtt{do}_A^{\mathrm{put}(b,c)}(x,y)_* \vee \mathrm{clear}(d,y)$
[N_1]	(6)	$\neg\mathrm{clear}(d,x) \vee \neg\mathtt{do}_E^{\mathrm{put}(c,d)}(x,y)_* \vee \mathrm{clear}(d,y)$
[N_2]	(7)	$\neg\mathrm{on}(b,c,x) \vee \neg\mathtt{do}_E^{\mathrm{put}(c,d)}(x,y)_* \vee \mathrm{on}(b,c,y)$
[C_1]	(8)	$\neg\mathrm{on}(b,c,x) \vee \neg\mathrm{on}(c,d,x) \vee \neg\mathrm{clear}(d,x) \vee \mathrm{tower}(b,c,d,x)_*$
[K_1]	(9)	$q_{\mathbf{K}_E\langle\mathtt{do}_E(\mathrm{put}(b,c))\rangle\top}(\epsilon)$
[K_1]	(10)	$\neg q_{\mathbf{K}_E\Diamond_E\mathrm{tower}(b,c,d)}(\epsilon)$
[K_1]	(11)	$q_{\mathbf{K}_E\Diamond_E\mathrm{tower}(b,c,d)}(x) \vee K_E(x, h_{K_E}(x))_*$
[K_1]	(12)	$q_{\mathbf{K}_E\Diamond_E\mathrm{tower}(b,c,d)}(x) \vee \neg\mathrm{tower}(b,c,d,y)_*$
[Ax]	(13)	$\neg q_{\mathbf{K}_E\langle\mathtt{do}_E(\mathrm{put}(b,c))\rangle\top}(x) \vee \neg\mathbf{K}_E(x,y)_* \vee q_{\langle\mathtt{do}_E(\mathrm{put}(b,c))\rangle\top}(y)$
[Ax]	(14)	$\neg q_{\langle\mathtt{do}_E(\mathrm{put}(b,c))\rangle\top}(x) \vee \mathtt{do}_A^{\mathrm{put}(b,c)}(x, g^c_b(x))_*$
[Ax]	(15)	$\neg q_{\mathbf{K}_E\mathrm{clear}(c)}(x) \vee \neg\mathbf{K}_E(x,y)_* \vee \mathrm{clear}(c,y)$
[Ax]	(16)	$\neg q_{\mathbf{K}_E\mathrm{clear}(d)}(x) \vee \neg\mathbf{K}_E(x,y)_* \vee \mathrm{clear}(d,y)$

[I_1] (17) $q_{\mathbf{K}_E \text{clear}(d)}(\epsilon)$
[I_1] (18) $q_{\mathbf{K}_E \text{clear}(d)}(\epsilon)$

We have obtained the refutation of $\text{CL}_{\text{DL}^*}(\Pi(\psi))$ by using the first-order theorem prover SPASS 1.0.0 [24] which implements the resolution framework of Bachmair and Ganzinger [1]. As an ordering we used a recursive path ordering. Since any recursive path ordering is compatible with the strict subterm ordering, SPASS is a decision procedure by Theorem 3. In every non-unit clause we marked the maximal literal of the clause by an index $\cdot_*$. Thus, inference steps are restricted to these literals.

We observe that clause (12) consists of two variable-disjoint subclauses. This clause will be subject to splitting which introduces two branches into our search space: One on which the unit clause $q_{\mathbf{K}_E \Diamond_E \text{tower}(b,c,d)}(x)$ is an element of the clause set and one on which the unit clause $\neg\text{tower}(b, c, d, y)$ is an element of the clause set instead. For the first set of clauses we directly obtain a contradiction using clause (10). For the second set of clauses

[12.2] (19) $\neg\text{tower}(b, c, d, y)_*$

replaces clause (12). We see that among the clause (1) to (17), only (1), (8), (14), and (11) contain a positive literal which is maximal and can thus serve as positive premises in resolution steps. We can derive among others the following clauses.

[11.2,13.2] (20) $q_{\mathbf{K}_E \text{tower}(b,c,d)}(x) \vee \neg q_{\mathbf{K}_E \langle \texttt{do}_E(\text{put}(b,c))\rangle \top}(x)$
$\qquad \vee\, q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle \top}(h_{K_E}(x))_*$
[11.2,15.2] (21) $q_{\mathbf{K}_E \text{tower}(b,c,d)}(x) \vee \neg q_{\mathbf{K}_E \text{clear}(c)}(x) \vee \text{clear}(c, h_{K_E}(x))$
[11.2,16.2] (22) $q_{\mathbf{K}_E \text{tower}(b,c,d)}(x) \vee \neg q_{\mathbf{K}_E \text{clear}(d)}(x) \vee \text{clear}(d, h_{K_E}(x))$
[14.2, 2.2] (23) $\neg q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle \top}(x) \vee \text{on}(b, c, g^c_b(x))_*$
[14.2, 4.2] (24) $\neg\text{clear}(c, x) \vee \neg q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle \top}(x) \vee \text{clear}(c, g^c_b(x))_*$
[14.2, 5.2] (25) $\neg\text{clear}(d, x) \vee \neg q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle \top}(x) \vee \text{clear}(d, g^c_b(x))_*$
[1.3, 3.1] (26) $\neg\text{clear}(c, x) \vee \neg\text{clear}(d, x) \vee \text{on}(c, d, g^d_c(x))_*$
[1.3, 6.2] (27) $\neg\text{clear}(c, x) \vee \neg\text{clear}(d, x) \vee \text{clear}(d, g^d_c(x))_*$
[1.3, 7.2] (28) $\neg\text{clear}(c, x) \vee \neg\text{clear}(d, x) \vee \neg\text{on}(b, c, x) \vee \text{on}(b, c, g^d_c(x))_*$
[8.4,19.1] (29) $\neg\text{clear}(d, x) \vee \neg\text{on}(b, c, x) \vee \neg\text{on}(c, d, x)_*$

Intuitively, clause (29) says that there is no situation x in which the blocks b, c, and d form a tower. The remainder of the derivation shows that this assumption leads to a contradiction. We choose clause (26) to derive the following clause.

[26.3,29.3] (30) $\neg\text{clear}(c, x) \vee \neg\text{clear}(d, x)$
$\qquad \vee\, \neg\text{clear}(d, g^d_c(x)) \vee \neg\text{on}(b, c, g^d_c(x))_*$

Note that in clause (30) all literals containing a Skolem term originate from the negative premise (29) while all the remaining literals originate from the positive premise (26). Intuitively, literals containing the Skolem term $g^d_c(x)$ impose constraints on the situation we are in after performing a $\text{put}(c, d)$ action in a situation x, while the remaining literals which have x as their final argument impose constraints on situation x itself.

Since literals containing a Skolem term are deeper than the remaining literals, the ordering restrictions on the resolution inference rule restrict applications of resolution to these literals. In the following part of the derivation we consecutively eliminate these literals by resolution inferences with the clauses (27) and (28) and obtain

$$(31) \quad \neg\text{clear}(c,x) \vee \neg\text{clear}(d,x) \vee \neg\text{on}(b,c,x)_*$$

Here the literal $\neg\text{on}(b,c,x)$ is maximal and we choose clause (23) which is related to a put(b,c) action as positive premise.

[23.2,31.4] $(32) \quad \neg q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle\top}(x) \vee \neg\text{clear}(c, g_b^c(x)) \vee \neg\text{clear}(d, g_b^c(x))$

By inference steps with the clauses (24) and (25) we eliminate all literals containing Skolem terms and obtain

$$(33) \quad \neg q_{\langle \texttt{do}_E(\text{put}(b,c))\rangle\top}(x) \vee \neg\text{clear}(c,x) \vee \neg\text{clear}(d,x)$$

Using clause (20), (21), and (22) we obtain

[11.2,15.2] $(34) \quad q_{\mathbf{K}_E\text{tower}(b,c,d)}(x) \vee \neg q_{\mathbf{K}_E\langle \texttt{do}_E(\text{put}(b,c))\rangle\top}(x) \vee \neg q_{\mathbf{K}_E\text{clear}(c)}(x) \vee \neg q_{\mathbf{K}_E\text{clear}(d)}(x)$

from which we can finally derive a contradiction by (10), (9), (17), and (18).

Solving the Eve Example By Temporal Resolution. The specification of the problem can be written as formulae in the normal form as follows. For example (E_1) instantiated where $X = c$ and $Y = d$ can be written as the following two rules.

$$\mathbf{true} \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_E(\text{put}(c,d)) \vee \text{on}(c,d))$$
$$\mathbf{true} \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_E(\text{put}(c,d)) \vee \neg\text{clear}(c))$$

The conjunction of initial conditions is rewritten by a new proposition v and the conjuncts $\mathbf{K}_E\text{clear}(c)$ and $\mathbf{K}_E\text{clear}(d)$ can be can be written as follows

(I_2) $\quad \mathbf{start} \rightarrow v$

(I_3) $\quad \mathbf{true} \rightarrow \neg v \vee \mathbf{K}_E\text{clear}(c)$

(I_4) $\quad \mathbf{true} \rightarrow \neg v \vee \mathbf{K}_E\text{clear}(d)$

Translating (K_1) into the CTL formulation we obtain

$$\mathbf{K}_E\mathbf{E}\bigcirc(\text{done}_A(\text{put}(b,c))) \wedge \neg\mathbf{K}_E\mathbf{E}\diamond\text{tower}(b,c,d).$$

Next we rewrite into the normal form introducing new propositions w, x, y, z and replacing tower(b,c,d) with its definition. Thus w replaces the above conjunction (G_1), y replaces the subformula $\mathbf{E}\bigcirc(\text{done}_A(\text{put}(b,c)))$ (G_2), (G_3), and z replaces $\neg\mathbf{E}\diamond\text{tower}(b,c,d)$ (G_4) which is equivalent to $\mathbf{A}\square\neg\text{tower}(b,c,d)$. The latter formula is rewritten into SNF$_{karo}$ using the new proposition x (G_5), (G_6), (G_7).

(G_1) $\quad \mathbf{start} \rightarrow w$

(G_2) $\quad \mathbf{true} \rightarrow \neg w \vee \mathbf{K}_E y$

(G_3) $\quad y \rightarrow \mathbf{E}\bigcirc(\text{done}_A(\text{put}(b,c)))$

(G_4) $\quad \mathbf{true} \rightarrow \neg w \vee \neg\mathbf{K}_E \neg z$

(G_5) $\quad \mathbf{true} \rightarrow \neg z \vee x$

(G_6) $\quad x \rightarrow \mathbf{A}\bigcirc x$

(G_7) $\quad \mathbf{true} \rightarrow \neg x \vee \neg\text{on}(b,c) \vee \neg\text{on}(c,d) \vee \neg\text{clear}(d)$

Firstly, we apply the rules SRES2 and SRES5 to (G_6), (G_7), and instantiations of (N_1), (N_2), (E_1), and (A_1) given below

(N_1) $\quad \text{clear}(d) \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_E(\text{put}(c,d)) \vee \text{clear}(d))$

(N_2) $\quad \text{on}(b,c) \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_E(\text{put}(c,d)) \vee \text{on}(b,c))$

(E_1) $\quad \mathbf{true} \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_E(\text{put}(c,d)) \vee \text{on}(c,d))$

(A_1) $\quad \text{clear}(c) \wedge \text{clear}(d) \rightarrow \mathbf{E}\bigcirc\text{done}_E(\text{put}(c,d))_{\langle \mathsf{c}_1 \rangle}$

obtaining

$$x \wedge \text{clear}(d) \wedge \text{on}(b,c) \wedge \text{clear}(c) \rightarrow \mathbf{E}\bigcirc\mathbf{false}_{\langle \mathsf{c}_1 \rangle}.$$

An application of SRES4 to this step clause results in

(G_8) $\quad \mathbf{true} \rightarrow \neg x \vee \neg\text{clear}(d) \vee \neg\text{on}(b,c) \vee \neg\text{clear}(c).$

Next we again apply the rules SRES2, and SRES5 to (G_6), (G_8), (G_3) and the following instantiations of (N_1) and (E_1)

(N_1) $\quad \text{clear}(c) \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_A(\text{put}(b,c)) \vee \text{clear}(c))$

(N_1) $\quad \text{clear}(d) \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_A(\text{put}(b,c)) \vee \text{clear}(d))$

(E_1) $\quad \mathbf{true} \rightarrow \mathbf{A}\bigcirc(\neg\text{done}_A(\text{put}(b,c)) \vee \text{on}(b,c))$

obtaining

$$\text{clear}(c) \wedge \text{clear}(d) \wedge x \wedge y \rightarrow \mathbf{E}\bigcirc\mathbf{false}_{\langle \mathsf{c}_2 \rangle}.$$

With an application of SRES4 to this clause we obtain

(G_9) $\quad \mathbf{true} \rightarrow \neg x \vee \neg y \vee \neg\text{clear}(c) \vee \neg\text{clear}(d)$

and then resolving with (G_5) using KRES1 we derive the following.

(G_{10}) $\quad \mathbf{true} \rightarrow \neg z \vee \neg y \vee \neg\text{clear}(c) \vee \neg\text{clear}(d)$

(G_{10}) is then resolved with (G_4) using KRES4 to obtain

(G_{11}) $\quad \mathbf{true} \rightarrow \neg w \vee \neg\mathbf{K}_E y \vee \neg\mathbf{K}_E\text{clear}(c) \vee \neg\mathbf{K}_E\text{clear}(d)$

which can be resolved with the initial conditions (I_3), (I_4), and (G_2) using KRES1 to obtain

(G_{12}) $\quad \mathbf{true} \rightarrow \neg w \vee \neg v.$

Finally resolving (G_{12}) with (I_2) and (G_1) using IRES1 the contradiction

$$\mathbf{start} \rightarrow \mathbf{false}$$

is obtained.

6 Conclusion

In this paper we have considered a fragment of the KARO framework of rational agency called the core KARO logic. We presented two approaches to providing practical proof methods for the core KARO logic, namely, an approach based on a combination of a translation of formulae in the core KARO logic to first-order logic and theorem proving by an ordering refined resolution calculus, and an approach based on a combination of an embedding of the core KARO logic into the fusion of CTL and $\mathsf{S5}_{(m)}$, a normal form transformation for this logic, and theorem proving by a non-classical resolution calculus.

Both approaches are able to provide sound, complete, and terminating proof methods for the core KARO logic excluding the implementability operator. We have discussed the problems with the implementability operator with respect to both approaches and suggested a possible solution in one of the approaches.

A comparion of the two approaches shows that the translation approach allows to deal quite elegantly with the informational component of KARO while the clausal temporal resolution approach has a better potential to provide a complete calculus for the dynamic component of KARO, in particular, in the presence of unbounded repetition. We believe that a detailed analysis of both approaches will help us to overcome their respective limitations. We also hope that it will help us to improve the practicality of both methods, by pointing out redundancies present in the proof search of either method.

An alternative approach is to consider the combination of both approaches taking their respective strength into account. In [14] we present a combination of clausal temporal resolution (restricted to a linear time temporal logic) and the translation approach plus first-order resolution (restricted to extensions of the multi-modal logic $\mathsf{K}_{(m)}$), and we were able to show soundness, completeness, and termination of this combination for a range of combined logics. We are confident that we can extend this combination to cover the fusion of CTL and various modal logics plus additional operators like implementability, which would bring us closer to our goal of providing proof methods for the core KARO logic and beyond.

References

1. L. Bachmair and H. Ganzinger. Resolution theorem proving. To appear in J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*.
2. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems (extended abstract). In *Proc. ATAL-98*, volume 1555 of *LNAI*. Springer, 1999.
3. A. Bolotov and M. Fisher. A clausal resolution method for ctl branching time temporal logic. *Journal of Experimental and Theoretical Artificial Intelligence*, 11:77–93, 1999.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.
5. H. de Nivelle. Translation of S4 into GF and 2VAR. Manuscript, May 1999.

6. H. De Nivelle, R. A. Schmidt, and U. Hustadt. Resolution-based methods for modal logics. *Logic Journal of the IGPL*, 8(3):265–292, 2000.
7. C. Dixon, M. Fisher, and A. Bolotov. Resolution in a Logic of Rational Agency. In *Proc. ECAI 2000*, pages 358–362. IOS Press, 2000.
8. C. Dixon, M. Fisher, and M. Wooldridge. Resolution for temporal logics of knowledge. *Journal of Logic and Computation*, 8(3):345–372, 1998.
9. C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Method for the Decicion Problem*, volume 679 of *LNCS*. Springer, 1993.
10. M. Fisher, C. Dixon, and M. Peim. Clausal Temporal Resolution. ACM Transactions on Computational Logic, 2(1), 2001.
11. R. Goré. Tableau methods for modal and temporal logics. In M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Kluwer, 1999.
12. J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time I: Lower bounds. *Journal of Computer and System Sciences*, 38:195–237, 1989.
13. U. Hustadt. *Resolution-Based Decision Procedures for Subclasses of First-Order Logic*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1999.
14. U. Hustadt, C. Dixon, R. A. Schmidt, and M. Fisher. Normal forms and proofs in combined modal and temporal logics. In *Proc. FroCoS'2000*, volume 1794 of *LNAI*, pages 73–87. Springer, 2000.
15. U. Hustadt and R. A. Schmidt. Using resolution for testing modal satisfiability and building models. To appear in the *Journal of Automated Reasoning*, 2001.
16. B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Formalizing abilities and opportunities of agents. *Fundamenta Informaticae*, 34(1,2):53–101, 1998.
17. G. Mints. Gentzen-type systems and resolution rules. Part I: Propositional logic. In *Proc. COLOG-88*, volume 417 of *LNCS*, pages 198–231. Springer, 1990.
18. H. J. Ohlbach. Combining Hilbert style and semantic reasoning in a resolution framework. In *Proc. CADE-15*, volume 1421 of *LNAI*, pages 205–219. Springer, 1998.
19. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
20. A. S. Rao and M. P. Georgeff. Modeling agents withing a BDI-architecture. In *Proc. KR-91*, pages 473–484. Morgan Kaufmann, 1991.
21. R. A. Schmidt. Decidability by resolution for propositional modal logics. *Journal of Automated Reasoning*, 22(4):379–396, 1999.
22. B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Communicating rational agents. In *Proc. KI-94*, volume 861 of *LNAI*, pages 202–213. Springer, 1994.
23. B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. How to motivate your agents. In *Intelligent Agents II*, volume 1037 of *LNAI*. Springer, 1996.
24. C. Weidenbach et al. System description: SPASS version 1.0.0. In *Proc. CADE-16*, volume 1632 of *LNAI*, pages 378–382. Springer, 1999.
25. G. Weiß, editor. *Multiagent systems: A modern approach to distributed artificial intelligence*. MIT Press, 1999.

On Formal Modeling of Agent Computations

Tadashi Araragi[1], Paul Attie[2,3], Idit Keidar[2], Kiyoshi Kogure[1],
Victor Luchangco[2], Nancy Lynch[2], and Ken Mano[1]

[1] NTT Communication Science Laboratories, 2-4 Hikaridai Seika-cho Soraku-gun,
Kyoto 619-0237 Japan.
{araragi, kogure, mano}@cslab.kecl.ntt.co.jp

[2] MIT Laboratory for Computer Science, 545 Technology Square,
Cambridge, MA, 02139, USA.
{idish, victor_l, lynch}@theory.lcs.mit.edu

[3] College of Computer Science, Northeastern University, Cullinane Hall,
360 Huntington Avenue, Boston, Massachusetts 02115.
attie@ccs.neu.edu

1 Introduction

This paper describes a comparative study of three formal methods for modeling and validating agent systems. The study is part of a joint project by researchers in MIT's Theory of Distributed Systems research group and NTT's Cooperative Computing research group. Our goal is to establish a mathematical and linguistic foundation for describing and reasoning about agent-style systems.

Agents are autonomous software entities that cooperate with other agents in carrying out delegated tasks. A key feature of agent systems is that they are dynamic in that they allow run-time creation and destruction of processes, run-time modification of communication capabilities, and mobility.

Formal models are needed for careful description and reasoning about agents, just as for other kinds of distributed systems. Important issues that arise in modeling agent computation are: agent communication, dynamic creation and destruction of agents, mobility, and naming. Our project examines the power of the following three formal methods in studying different agent applications:

1. Erdös [1] is a knowledge-based environment for agent programming; it supports dynamic creation of agents, dynamic loading of programs, and migration. Erdös also supports automatic model checking using CTL [4].
2. Nepi^2 [6] is a programming language for agent systems, based on the π-calculus [8]. It extends the π-calculus with data types and a facility for communication with the environment.
3. I/O automata [7] is a mathematical framework extensively used for modeling and reasoning about systems with interacting components. Prior to this project, I/O automata were not used for reasoning about agents. As part of this project, we have extended the I/O automata formalism to account for issues that arise in agent systems, such as dynamic creation and destruction of agents, mobility, and naming (see [3]).

We model a simple e-commerce system using the three formalisms. In Section 2 we describe the e-commerce system. We present the different agents that comprise the system, and their interaction. In the following sections, we present our modeling of two of the system components using the three formalisms. For lack of space, we do not include models of the other system components, specifications or correctness proofs. In [3] we present a variant of the example, with all the system components, a specification, and a correctness proof.

Sections 3, 4, and 5 discuss I/O automata, Nepi^2, and Erdös, respectively. Each section begins with a presentation of the specific formalism, in particular, describing how agent communication, dynamic creation and destruction, mobility, and naming are modeled. Each of these sections then continues to present the e-commerce example in the specific formalism. Finally, each section discusses the specification and verification styles used with the specific formalism.

Beyond examining the individual formalisms, we also strive to achieve cross-fertilization among them: we aim to enrich each of the formalisms with concepts from the others. For example, the mathematical framework of I/O automata may serve as a semantic model for a language like Erdös, which is appropriate for agent-style programming due to its knowledge-based style and standard agent communication interaction. In Section 6, we show a generic transformation from Erdös to I/O automata. In Section 7, we compare the three formalisms.

2 An Electronic Commerce Example

We consider a simple flight purchase problem, in which a client requests to purchase a ticket for a particular flight, given by some "flight information" f, and specifies a particular maximum price *mp* that the client is willing to pay. The request goes to a static (always existing) "client agent," which creates a special "request agent" dedicated to that particular request. The request agent communicates with a static "directory agent" to discover a set of databases where the request might be satisfied. Then the request agent communicates with some or all of those databases. Each active database has a front end, the database agent, which, when it receives such a communication, creates a special "response agent" dedicated to the particular client request. The response agent, consulting the database, tells the request agent about a price for which it is willing to sell a ticket for the flight. The response agent does not send information about flights that cost more than the maximum price specified in the request.

If the request agent has received at least one response with price no greater than *mp*, it chooses some such response. It then communicates with the response agent of the selected response to make the purchase. The response agent then communicates with the database agent to purchase the flight, and sends a positive confirmation if it is able to do so. If it cannot purchase the flight, it sends the request agent a negative confirmation.

Once the request agent has received a positive confirmation, it returns the information about the purchase to the client agent, which returns it to the client. Rather than attempting indefinitely to get a positive confirmation, the request agent may return a negative response to the client agent once it has received at least one negative response from each database that it knows about.

After the request agent returns the purchase information to the client, it sends a "done" message to all the database agents that it initially queried. This causes each database agent to destroy the response agent it had created to handle the client request. The request agent then terminates itself.

In this paper we present models for the client and request agent components, using the three formalisms. A fuller version of this paper [2] contains all five of the system components. We have not modeled the client formally; we do not need to, because we do not impose any restrictions on its submission of requests. So far, we have not formulated specifications for properties to be satisfied by the system. An example of such a property is: if (f, price) is returned to the client, then (f, price) actually exists in some database, $\mathit{price} \leq \mathit{mp}$, and a seat on the flight is reserved in the database.

We have formulated all the system components using the three formalisms. Due to space limitations, we include in this paper models only for the Client Agent and for the Request Agent. Note that in this example, agents are not mobile. In [3] we present a variant of this example, in which agents are mobile.

3 I/O Automata

3.1 Formal Foundation and the Modeling of Agents

The I/O automaton model [7] is a mathematical model for reactive systems and their components. An I/O automaton is a nondeterministic labeled transition system, together with an action signature. An automaton in a state s can execute an action a and move to a new state s' iff the triple (s, a, s') is in its transition relation. (We say that a is enabled in s.) The action signature partitions the actions into input, output, and internal actions. We augment the basic I/O automaton model to add support for dynamic behavior, including process creating and destruction, changing interfaces and mobility [3].

Agent communication. Automata communicate by executing common actions: an output action of an automaton can also be an input action of one or more other automata. No action can be an output action of two or more automata. Input actions must be enabled in every state of the automaton, i.e., inputs must always be accepted. An internal action of an automaton cannot be an action of any other automaton, so internal actions cannot be used to communicate. The input and output actions are *externally visible.* The external interface of an I/O automaton is the externally visible part of its action signature.

Dynamic creation and destruction. In addition to the "regular" actions of the I/O automaton model, we have create and destroy actions. A create action adds the specified automaton to the current set of automata that are considered "alive,"; a destroy action removes it. The current "global state" of a system, called a configuration, is a finite multiset of (automaton, local-state) pairs, with one pair for each alive automaton, and where the local-state component gives the current local state of the automaton. There is no general restriction on who may create or destroy an automaton, or on when this may happen.

Mobility. In our dynamic extension, automata may change their action signatures; a state transition may result in new sets of input, output, and internal actions for the automaton. The current action signature must always be contained within a fixed "universal" signature. Moreover, any input action in the signature corresponding to a state must be enabled in that state. Since the external interface is just the external part of the action signature, we can also dynamically change the external interface. We use this feature to model mobility, for example, in [3]. We model a system as consisting of a set of "logical" locations, and each agent (automaton) has a current location. In addition, we can use an automaton to model a channel between two locations (which allows us to model asynchrony, timing, message loss and/or reordering, etc.). An agent can interact directly with co-located agents or with an endpoint of a channel that is at the same location.

Naming. Naming is handled by assigning an identifier to each automaton. We usually require that, in any configuration of a system, all automata have unique identifiers. In this case, we say that the system is "clone-free."

3.2 The Client Agent

Client Agent: *ClAgt*

Universal Signature

Input:
  $\mathsf{request}(f, mp)$, where $f \in \mathcal{F}$ and $mp \in \Re^+$
  $\mathsf{req\text{-}agent\text{-}response}_r(f, mp, p, ok)$, where $r \in \mathcal{R}$, $f \in \mathcal{F}$, $mp, p \in \Re^+$, and $ok \in Bool$
Output:
  $\mathsf{response}(f, p, ok)$, where $f \in \mathcal{F}$, $p \in \Re^+$, and $ok \in Bool$
Internal:
  $\mathsf{create}(ClAgt, RqAgt_r(\langle f, mp\rangle))$, where $r \in \mathcal{R}$, $f \in \mathcal{F}$, and $mp \in \Re^+$

State

$reqs \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, outstanding requests; initially empty
$pends \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+$, outstanding requests for whom a request agent has been created, but the response has not yet returned to the client; initially empty
$resps \subseteq \mathcal{R} \times \mathcal{F} \times \Re^+ \times \Re^+ \times Bool$, responses not yet sent to client; initially empty
$created \subseteq \mathcal{R}$, indices of created request agents; initially empty

Transitions

```
In request(f, mp)
  Eff: $r \leftarrow choose(\mathcal{R} - created)$;
       $created \leftarrow created \cup \{r\}$;
       $reqs \leftarrow reqs \cup \{\langle r, f, mp\rangle\}$

Int create($ClAgt, RqAgt_r(\langle f, mp\rangle)$)
  Pre: $\langle r, f, mp\rangle \in reqs - pends$
  Eff: $pends \leftarrow pends \cup \{\langle r, f, mp\rangle\}$

In req-agent-response$_r(f, mp, p, ok)$
  Eff: $resps \leftarrow resps \cup \{\langle r, f, mp, p, ok\rangle\}$

Out response(f, p, ok)
  Pre: $\langle r, f, mp, p, ok\rangle \in resps$
  Eff: $reqs \leftarrow reqs - \{\langle r, f, mp\rangle\}$
       $pends \leftarrow pends - \{\langle r, f, mp\rangle\}$
       $resps \leftarrow resps - \{\langle r, f, mp, p, ok\rangle\}$
```

In the I/O automaton model of the client agent, the request input action (an output action of the "client environment", not modeled here) is parameterized by the flight f (of type $\mathcal{F}$) and the maximum price mp. $ClAgt$ assigns a unique identifier $r \in \mathcal{R}$ to each request, and adds the request to $reqs$. ($\mathcal{R}$ is an index set for the requests.) Subsequently $ClAgt$ creates a request agent $RqAgt_r(\langle f, mp\rangle)$ Note that the model handles two requests with the same f and mp as separate requests. The tuple $\langle r, f, mp\rangle$ is added to the set $pends$ of pending requests.

The $\mathsf{req\text{-}agent\text{-}response}_r$ input action models the receipt of a response from a request agent. The client agent adds the response to the set $resps$, and communicates the response to the client via the response output action. It also removes all record of the request at this point.

Since signatures do not vary in this example, we do not explicitly indicate their initial values; the signature is always equal to the universal signature.

3.3 The Request Agent

The request agent $RqAgt_r(\langle f, mp\rangle)$ handles the single request $\langle f, mp\rangle$, and then terminates itself. The $\mathsf{DIRquery}_r(f)$ and $\mathsf{DIRinform}_r(dbagents)$ actions model the interaction with the directory agent to find the set of active databases, which are indexed by $\mathcal{D}$. $RqAgt_r(\langle f, mp\rangle)$ queries these using $\mathsf{DBquery}_{r,d}(f, mp)$, and receives a response via the $\mathsf{RESPinform}_{d,r}(f, p)$ action. It attempts to buy a suitable flight via the $\mathsf{RESPbuy}_{r,d}(f, p)$ action, and receives a confirmation via the $\mathsf{RESPconf}_{d,r}(f, p, ok)$, which may be positive or negative, depending on the value of ok. It uses the $\mathsf{req\text{-}agent\text{-}response}(f, p, ok)$ output action, which is also an input action of $ClAgt$ as described above, to send the information to $ClAgt$.

The $\mathsf{DBdone}_{r,d}(f, mp)$ action tells a database agent that it can destroy the response agent that it created for that particular request. Finally, $RqAgt_r(\langle f, mp\rangle)$ destroys itself with the $\mathsf{destroy}(RqAgt_r(\langle f, mp\rangle))$ action. (The more general treatment in [3] also allows one automaton to destroy another one, in which case the destroy action is written with two arguments).

Request Agent: $RqAgt_r(f, mp)$ where $r \in \mathcal{R}$, $f \in \mathcal{F}$, and $mp \in \Re^+$

Universal Signature

Input:
- $\mathsf{DIRinform}_r(dbagents)$, where $dbagents \subseteq \mathcal{D}$
- $\mathsf{RESPinform}_{d,r}(f, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
- $\mathsf{RESPconf}_{d,r}(f, p, ok)$, where $d \in \mathcal{D}$, $p \in \Re^+$ and $ok \in Bool$

Output:
- $\mathsf{DIRquery}_r(f)$
- $\mathsf{DBquery}_{r,d}(f, mp)$, where $d \in \mathcal{D}$
- $\mathsf{RESPbuy}_{r,d}(f, p)$, where $d \in \mathcal{D}$ and $p \in \Re^+$
- $\mathsf{req\text{-}agent\text{-}response}(f, p, ok)$, where $p \in \Re^+$ and $ok \in Bool$
- $\mathsf{DBdone}_{r,d}$, where $d \in \mathcal{D}$

Internal:
- $\mathsf{terminate}(RqAgt_r(\langle f, mp\rangle))$

State

$resp \in (\mathcal{F} \times \Re^+ \times Bool) \cup \{\bot\}$, flight purchased but not yet sent to client; initially $\bot$
$localDB \subseteq \mathcal{D} \times \mathcal{F} \times \Re^+$, flights known with price at most mp; initially empty
$DBagts \subseteq \mathcal{D}$, known database agents; initially empty
$DBagtsleft \subseteq \mathcal{D} \cup \{\bot\}$, known database agents whose response agent has not yet returned a negative response; initially $\bot$
$bflag \in Bool$, boolean flag, *true* iff an attempt to purchase a ticket is in progress, initially *false*
$dflag \in Bool$, boolean flag, *true* iff a response has been returned to the client agent, initially *false*

Transitions

```
Out DIRquery_r(f)
  Pre: $true$

In DIRinform_r(dbagents)
  Eff: $DBagts \leftarrow dbagents$
       $DBagtsleft \leftarrow dbagents$

Out DBquery_{r,d}(f, mp)
  Pre: $d \in DBagts$

In RESPinform_{d,r}(f, p)
  Eff: $localDB \leftarrow localDB \cup \{\langle d, f, p\rangle\}$

Out RESPbuy_{r,d}(f, p)
  Pre: $\langle d, f, p\rangle \in localDB \wedge \neg bflag$
  Eff: $bflag \leftarrow true$

In RESPconf_{d,r}(f, p, ok)
  Eff: if $ok$ then
           $resp \leftarrow \langle f, p, ok\rangle$
       else
           $localDB \leftarrow localDB - \{\langle d, f, p\rangle\}$
           $DBagtsleft \leftarrow DBagtsleft - \{d\}$
           $bflag \leftarrow false$

Out req-agent-response(f, p, ok)
  Pre: $(resp = \langle f, p, ok\rangle \wedge ok) \vee$
       $(DBagtsleft = \emptyset \wedge \neg ok)$
  Eff: $dflag \leftarrow true$

Out DBdone_{r,d}(f, mp)
  Pre: $dflag \wedge d \in DBagts$
  Eff: $DBagts \leftarrow DBagts - \{d\}$

Int destroy(RqAgt_r(⟨f, mp⟩))
  Pre: $dflag \wedge DBagts = \emptyset$
```

3.4 Specification and Verification Style

The external behavior of an I/O automaton is described using *traces*, i.e., sequences of externally visible actions. The model includes good support for refinement and levels of abstraction. Correctness is defined in terms of *trace inclusion*: if every trace of an implementation is also a trace of the specification, then we consider the implementation to be a correct refinement of the specification. Trace inclusion is verified by establishing a *simulation relation* from the implementation to the specification, i.e., by showing that every transition of the implementation can be matched by a trace-equivalent sequence of transitions of the specification. I/O automata are usually described in a simple guarded command (precondition/effect) style, which we use here.

4 Erdös – An Internet Mobile Agents System

4.1 Formal Foundation and the Modeling of Agents

Erdös [1] is a knowledge-based environment for agent programming; it supports functions such as dynamic creation of agents, dynamic loading of programs, and migration. Erdös also supports automatic model checking using CTL [4].

An agent in Erdös consists of an agent program, a program stack, and a knowledge base. The knowledge base consists of logical formulae. The program consists of subprograms. The a subprogram "main" indicates the execution entry point. Subprograms are sequences of "test-actions" clauses (as in [5]), which are executed from head to tail, and take the following form:

$$\text{If } \mathit{test} \text{ then } \mathit{action}_1,\ ...,\ \mathit{action}_m \text{ else } \mathit{action}_{m+1},\ ...,\ \mathit{action}_n;$$

The *test* is a logical formula. In an execution of a "test-actions" clause, if the *test* can be deduced from the current knowledge base of the agent, then the agent executes action_1, ..., action_m; otherwise, it executes action_{m+1}, ..., action_n. A subprogram can call another subprogram using the *call* action, which specifies the name of the agent that holds the subprogram; an agent uses the constant "self" to specify its own name. The program stack maintains the execution state of the agent program, and it allows the execution to continue after migration.

We now discuss different aspects of modeling agents with Erdös.

Agent communication. Agents communicate by adding information to other each-other's knowledge bases. Specifically, the action $\mathit{add}(\mathit{agt\_name}{:}\ \phi)$ adds a formula ϕ to the knowledge base of the agent *agt_name*. An agent can remove a formula ϕ from its own knowledge base using the action $\mathit{rm}(\phi)$.

Dynamic creation and destruction. Agents are created using the $\mathit{create}(\mathit{agt\_name}{:}\ \mathit{new\_agt\_name},\ \mathit{sub\_prg\_name},\ \mathit{arg}1,\ ..)$ action. This action creates a new agent whose name is *new_agt_name*; the "main" subprogram of the new agent is *sub_prg_name*, and its program consists of all the subprograms recursively called from this "main" subprogram. The arguments $\mathit{arg}1$,... comprise the knowledge base of the new agent. Agents are destroyed using the $\mathit{kill}(\mathit{agt\_name})$ action, and can be stopped and resumed using the $\mathit{stop}(\mathit{agt\_name})$ and $\mathit{resume}(\mathit{agt\_name})$ actions.

Mobility. An agent can migrate to the location *url* using the $\mathit{go}(\mathit{url})$ action.

Naming. Erdös uses global agent names; the consistency of agent names is managed using a name server.

Interaction with the environment. In Erdös, the environment is thought as agents with Erdös agents' interface, That is, they exchange formulas with Erdös agents. The difference is that the environment agents are not programmed in Erdös.

4.2 The Client Agent

We present the Client Agent and the Request Agent using Erdös. In this example, ?mp, ?r, ... are variables instantiated in the test procedure and substituted over the "test-action" line. Agent and their names are dynamically created; these names are passed between agents. A trailing "-" after a test formula ϕ indicates that ϕ is to be removed from the knowledge base if the test succeeds. The return values of the external methods *ex_call* and *ex_call_i* ($i \in N$) are placed in the knowledge base as the formulas *Return*(...) and *Return_i*(...), respectively.

This code implements the specification of IOA as faithfully as possible. Of course, we have to add some controls of execution by actions such as call and idle. The only nondeterminism of Erdös agents is the matching of the test part.

Preconditions of IOA action may include some operations on data and, in Erdös, these operations must be done by ex_call actions before the associated knowledge test. For example, the data operations in the precondition of the create action of the IOA specification are executed in (1), (2) and (3). Here, pop_reqs method takes out an element from the set reqs and is_in_pend method checks $\neg\exists r' : \langle r', f, mp\rangle \in pends$.

```
sub_p main()
{if Request(?f,?mp)- then ex_call(add_reqs, Ele(?f,?mp));
 if true then ex_call_1(pop_reqs);                                  (1)
 if Return_1(?f,?mp) then ex_call_2(is_in_pends, Ele(-,?f,?mp));   (2)
 if Return_2(no) then ex_call_3(pop_R-created));                    (3)
 if Return_3(?r)- and Return_2(no)- and Return_1(?f,?mp)- then
     create(: ?r,req_agt_prg,Arg1(self),Arg2(?f),Arg3(?mp)),
     ex_call(add_pends, Ele(?r,?f,?mp)),
     ex_call(add_created,Ele(?r));
 if Req-agent-response(?r,?f,?mp,?p,?ok)- then
     ex_call(add_resps, Ele(?r,?f,?mp,?p,?ok));
 if true then ex_call(pop_resps);
 if Return(?r,?f,?mp,?p,?ok) then
     add(user: Response(?f,?p,?ok)),
     ex_call(rm_pends, Ele(?r,?f,?mp));
 if true then call(: main);
}
```

4.3 The Request Agent

In (1), (2) and (3), an idle action and call actions are used to control the execution. We also make copy DBagts* of DBagts for the control of avoiding the duplication of DBquery. We omit the code of the subprogram broad_cast_done_to_dbagts.

```
sub_p main()
{ if true then
      add(dir_agt: DIRquery(self,?f)),
      add(: Not-buy);
  if DIRinform(?dbagents) then
      ex_call(set_DBagts, Set(?dbagents)),
      ex_call(set_DBagtsleft, Set(?dbagents)),
      ex_call(set_DBagts*, Set(?dbagents));
  else idle;

 if true then call(: business);
}

sub_p business()
{
 if true then ex_call(pop_DBagts*);
 if Return(?d)- and Arg1(?f) and Arg2(?mp) then add(?d: DBquery(self,?f,?mp));
 if RESPinf(?resp_agt,?d,?f,?p) then ex_call(add_localDB,Ele(?d,?f,?p));
 if true then ex_call(get_localDB);
 if Return(?d,?f,?p)- and Not-bflg and RESPinf(?resp_agt,?d,?f,?p) then
     add(?resp_agt: RESbuy(self,?f,?p)), ex_call(pop_localDB, Ele(?d,?f,?p)),
     add(: Bflag), rm(: RESPinf(?resp_agt,?d,?f,?p));
 if RESPconf(?resp_agt,?d,?f,?p,?ok?) then else idle;                 (1)
 if RESPconf(?resp_agt,?d,?f,?p,true) then add(: Resp(?f,?p,true));
 if RESPconf(?resp_agt,?d,?f,?p,false) then
     ex_call(rm_DBagtsleft, Ele(?d)),
     rm(: Bflag), add(: Not-bflag);
 if Resp(?f,?p,true)- and Arg1(?clt_agt) then
     add(?clt_agt: Req-agent-response(self,?f,?mp,?p,true)),
     add(: Dflag), call(: broad_cast_done_to_dbagts);                 (2)
 if true then ex_call(is_empty_DBagtsleft);
 if Return(yes)- and Arg1(?clt_agt) then
     add(?clt_agt: Req-agent-response(self,?f,?mp,?p,false)),
     add(: Dflag), call(: broad_cast_done_to_dbagts);
 if true then call(: business);                                       (3)
}
```

4.4 Specification and Verification Style

Erdös programs can be validated using CTL model checking: Erdös transforms a set of agent programs to a boolean formula that expresses the transition relation of the possible asynchronous behaviors of the agents. CTL model checking can then be applied to this formula.

The transition relation is expressed using the state variables $agt_k.st_i$, $agt_k.o_j$, run_k and $alive_k$. $agt_k.st_i$ expresses the fact that the state of execution (i.e., the program stack) at agent agt_k is currently st_i. The variable $agt_k.o_j$ expresses the fact that the formula with id f_j currently occurs in the knowledge base of agent agt_k. run_k expresses the fact that currently it is agt_k's turn to run, and $alive_k$ expresses the fact that agt_k is alive.

The test formula in a “test-actions” clause of agt_k is the precondition of the actions in the same clause. The test formula is transformed to the condition of the boolean formula expressed by $agt_k.o_j$ using theorem proving techniques.

CTL model checking is restricted to finite state systems. We therefore have to extract a finite state abstraction of the system. To this end, we assume an upper bound on the depth of program stack (this bound results from static analysis of the agent program). We require programmers to manually abstract into finite infinite length data structures used in the program.

The asynchronous behavior is expressed by the interleaving model. More specifically, we impose boolean formulas $\bigwedge_{i<j} \neg(run_i \wedge run_j)$ and $\bigvee_i run_i$, to model the fact that each agent has one thread.

5 Nepi^2

5.1 Formal Foundation and the Modeling of Agents

Nepi^2 [6] is a programming language based on the π-calculus [8]. It extends the π-calculus with data types and a facility for communication with the environment.

Nepi^2 supports control primitives such as parallel composition of processes (`par` P_1 P_2), alternative choice ($+$ P_1 $\cdots$ P_n), conditional (`if` *condition* P_1 P_2), generation of a fresh channel `c` (`new c` P), channel input (? `c` ($\boldsymbol{x}$) P) and output (! `c` ($\boldsymbol{v}$) P). These primitives are derived from the π-calculus.

The semantics of Nepi^2 are described in the style of structural operational semantics. For instance, communication, recursion and parallel composition are defined by the following rules:
COMM: If $\mathtt{c}_1 = \mathtt{c}_2$, then

$$(\mathtt{par}\ (+\ \cdots\ (!\ \mathtt{c}_1\ (\boldsymbol{v})\ P)\ \cdots)\ (+\ \cdots\ (?\ \mathtt{c}_2\ (\boldsymbol{x})\ Q)\ \cdots)) \xrightarrow{\tau} (\mathtt{par}\ P\ Q[\boldsymbol{v}/\boldsymbol{x}]).$$

REC: If there is a process definition (`defproc` A ($\boldsymbol{x}$) P), then

$$(A\ \boldsymbol{v}) \xrightarrow{\tau} P[\boldsymbol{v}/\boldsymbol{x}].$$

PAR: For every action a,

$$\frac{P \xrightarrow{a} P'}{(\mathtt{par}\ P\ Q) \xrightarrow{a} (\mathtt{par}\ P'\ Q)}.$$

Agent communication. Communication in Nepi^2 is synchronous (also called *rendezvous-style*) because an output action and the corresponding input action occur simultaneously.

Dynamic creation and destruction. Agents can be dynamically created using the parallel composition operator. For example, a code fragment (`par` P_1 P_2) can cause the creation of a new agent P_1 in parallel with the invocation of the original agent's continuation, P_2.

Mobility. An agent is migrated by passing its code over a channel to a different location.

Naming. When a fresh channel is generated using `new` it is assigned with a new name. One or more fresh channels are given to a created agent as its identity.

Interaction with the environment. Nepi^2 is implemented in Lisp, and Lisp functions can be used for data operations in Nepi^2 programs. Communication with the environment is supported using Unix standard input and output.

5.2 The Client Agent

```
(defproc ClientAgent (request response dir req-agt-resp)
  (+
    (? request (fltinfMp)
      (par
        (new req (new reqconf
          (ReqAgtInit fltinfMp dir req-agt-resp req reqconf))
          (ClientAgent request response dir req-agt-resp) ))
     (? req-agt-resp (ok?FltinfP)
       (! response (ok?FltinfP)
         (ClientAgent request response dir req-agt-resp) ))))
```

A client agent ClientAgent gets four channels as arguments: request and response for communication with the client, dir of the directory agent, and req-agt-resp for communication with request agents. It chooses the following actions nondeterministically: receiving a tuple fltinfMp of the flight information and maximum price via request, or receiving a tuple ok?FltinfP of the flag, flight information and price via req-agt-resp. If the former action is chosen, then the primitive par creates a request agent and the continuation of the client agent. The created request agent is given two fresh channels req and reqconf as its identity.

5.3 The Request Agent

```
(defproc ReqAgtInit (fltinfMp dir req-agt-resp req reqconf)
  (! dir ((list req (car fltinfMp)))
    (? req (DBagents)
      (par (DBMulticast DBagents 'query req fltinfMp)
          (ReqAgt req-agt-resp req reqconf DBagents
                  (length DBagents) )))))

(defproc ReqAgt (req-agt-resp req reqconf DBagents DBagentsLeft)
  (? req (respFltinfP)
    (! (car respFltinfP) ((cons reqconf (cdr respFltinfP)))
      (? reqconf (ok?FltinfP)
        (if (car ok?FltinfP)
          (! req-agt-resp (ok?FltinfP)
            (DBMulticast DBagents 'done req fltinfMp) )
          (if (== DBagentsLeft 0)
            (! req-agt-resp (ok?FltinfP)
              (DBMulticast DBagents 'done req fltinfMp))
            (ReqAgt req-agt-resp req reqconf DBagents
                    (- DBagentsLeft 1) )))))))

(defproc DBMulticast (DBagents cmd req fltinfMp)
  (if (eq* DBagents nil)
    end
    (! (car DBagents) ((cons cmd (cons req fltinfMp)))
      (DBMulticast (cdr DBagents) cmd req fltinfMp) )))
```

A process ReqAgtInit queries the directory agent for the available database agents, multicasts a query about fltinfMp to them, and invokes ReqAgt. Upon receiving a response via req, ReqAgt send a 'buy' request for the response. Then, if it receives a positive confirmation via reqconf, it forwards the response to the client agent via `req-agt-resp`, and multicasts a 'done' message to all the available database agents. Otherwise, it repeats the same thing with `DBagentsLeft` decremented, or, if all the database agents responded negatively, it forwards the last negative response to the client agent.

5.4 Specification and Verification Style

Currently, there is no support for property specification and verification in Nepi^2. It should be straightforward to adapt these from the well-developed theory of the π-calculus, which uses verification methods based on bisimulation equivalence, and Hennessy-Milner logic extended by a fixed point operator.

6 From Erdös to I/O Automata

Erdös is a convenient language for knowledge-based programming of agent systems. We now show how the mathematical framework of I/O automata may serve as a semantic model for Erdös: we show a generic transformation from Erdös to an I/O automaton.

The transformation is not difficult – the computational semantics of Erdös have much common with I/O automata. Both formalisms employ state based control, and with both, input actions are always enabled. Below, we give an example of a generic transformation of an agent, agt_k, to an I/O automaton.

We denote by KB_{agt_k} the knowledge base of agt_k, and by F_{agt_k}, the set of formulae that may occur in KB_{agt_k}. *prg_stk* is the program stack of the agent. We denote the actions that appear in the n'th "test-actions" clause of the subprogram *subp* as: $\langle subp, n\rangle$. The test match(*prg_stk*, $\langle subp, n\rangle$) checks if the top of *prg_stk* points to $\langle subp, n\rangle$. modify(*prg_stk*, $\langle subp, n\rangle$) modifies the *prg_stk* according to the action $\langle subp, n\rangle$ In the automaton below, we only show the actions in the "then" clause. Those of the "else" clause are similar; the only difference is that $KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$ is replaced by $KB_{agt_k} \not\vdash \text{test}_{\langle subp,n\rangle}$.

Transitions

In $\mathsf{add}(agt_k : f) f \in F_{agt_k}$
  Eff: $KB_{agt_k} = KB_{agt_k} \cup \{f\}$

Out $\mathsf{add}(agt_j : \psi)$
  Pre: $\text{match}(prg\_stk, \langle subp, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $\text{modify}(prg\_stk, \langle subp, n\rangle, \text{none})$

Int $\mathsf{rm}(: \psi)$
  Pre: $\text{match}(prg\_stk, \langle subp, n\rangle) \wedge KB_{agt_k} \vdash test_{\langle subp,n\rangle}$
  Eff: $KB_{agt_k} = KB_{agt_k} - \{\psi\}$
    $\text{modify}(prg\_stk, \langle subp, n\rangle, none)$

Int $\mathsf{call}(agt_j : sub\_prg)$
  Pre: $\text{match}(prg\_stk, \langle subp, n\rangle) \wedge KB_{agt_k} \vdash \text{test}_{\langle subp,n\rangle}$
  Eff: $\text{modify}(prg\_stk, \langle subp, n\rangle, sub\_prg)$

Int idle
  Pre: $\mathrm{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \mathrm{test}_{\langle subp,n\rangle}$

Int create$(agt_j : agt_m, sub_prg, arg1, \dots)$
  Pre: $\mathrm{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \mathrm{test}_{\langle subp,n\rangle}$
  Eff: $\mathrm{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, \mathrm{none})$

Int ex_call$(method, arg)$
  Pre: $\mathrm{match}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle) \wedge KB_{agt_k} \vdash \mathrm{test}_{\langle subp,n\rangle}$
  Eff: $method(arg)$
    $\mathrm{modify}(\mathit{prg\_stk}, \langle \mathit{subp}, n\rangle, \mathrm{none})$

7 Discussion

We have presented three formalisms for modeling and reasoning about agent-style distributed systems; we now compare several aspects of these formalisms.

I/O automata is a mathematical model for modeling interacting components. The same mathematical model can be used at several levels of abstraction, from modeling of implementations to specifications. Advantages of the I/O automata approach include the fact that it supports compositional, invariant, and simulation proofs, including computer-assisted verification. In particular, the I/O automata model has a well established infinite-state verification method. Also, the allowance of nondeterminism is a big advantage for specifications. However, in contrast to knowledge-based programming and variants of π-calculus, there is little experience in using I/O automata to model dynamic systems. In our project, we are gaining such experience.

Erdös is an agent programming system/language. Erdös programs may be verified using CTL model checking. An advantage of Erdös is that it is suitable for knowledge-based programming and reasoning, as is often employed in agent systems. The knowledge-based style makes the program semantics easy to understand. Erdös also offers an automated verification facility. However, in contrast to I/O automata, Erdös' verification methods are limited to finite-state systems. Currently, we overcome this limitation by manually abstracting programs to a finite-state system. Using I/O automata as a semantic model for Erdös (as illustrated above) may present a solution to this limitation.

Nepi^2 is a network programming system. Its language and computation model are based on an extension of the π-calculus with data type. A major advantage of Nepi^2 is that a problem can be written concisely using π-calculus primitives, which have been used extensively for agent modeling. In particular, the fresh channel generation operator `new` is helpful for naming created agents. Currently, there is no support for property specification and verification in Nepi^2. It should not be difficult to adapt these from the well-developed theory of the π-calculus.

Communication and control. With both I/O automata and Erdös, an input action is always possible (enabled). In Nepi^2, on the other hand, communication is based on a handshake model: an input action is possible only when it is performed explicitly, and an output action can block the execution in the absence of corresponding input actions. Input-enabling lends itself naturally to programming in event driven style. It is more difficult to model event driven systems using handshake-based communication, especially if different system components (agents) are developed independently.

With I/O automata and Nepi^2, agents in different systems interact via different actions. In contrast, with Erdös agents in all systems interact via a generic interface of "add" actions.

Dynamic creation and naming. All three formalisms provide simple interfaces for creating new agents. The dynamic version of I/O automata uses indices to differentiate agents created by the same automaton. It is left to the programmers to maintain unique names for agents created by different automata. Upon creation of a new agent, Erdös assigns it a name consisting of the creating agent's name and a programmer specified name. Nepi^2 uses a name server which is part of the environment to produce new agent names. Thus, the uniqueness of names used with I/O automata and Erdös depends on the programmer, whereas with Nepi^2 it depends on the name server.

With all three formalisms, the name of a created agent may be passed as a parameter to other agents, to notify them of the new agent's existence. However, Erdös' verification model does not presently support passing of agent names.

Mobility. All three formalisms supports the mobility of agents in their computation models. In the dynamic version of I/O automata, mobility is modeled by changing the signature of an automaton. With Erdös and Nepi^2, on the other hand, mobility is modeled explicitly, and it does not change the semantics of computation. Of the three formalisms, Erdös is the only one that supports dynamic loading of programs in its computation model. This feature is useful for agents that need to adapt to unknown environments.

Specification and Verification. Being a mathematical model, properties of I/O automata can be specified in any sound mathematical form. In particular, properties are typically specified in one of two ways: either as trace properties formulated in logic, or as state machines that generate the set of *legal* traces. Specifications are usually formulated with a high level of nondeterminism. An algorithm is said to *implement* the specification if its traces are a *subset* of the traces of the specification. Both algorithms and specifications, in general, are infinite state.

For state machine style specifications, verification is done by showing a *simulation* from the algorithm to the specification. Simulations can be verified using interactive theorem provers.

In contrast to I/O automata, in the π-calculus, specifications are based on bi-simulation and algebraic equivalences. The verification methods used in the π-calculus emphasize bi-simulation equivalence, and Hennessy-Milner logic extended by a fixed point operator. With bi-simulation, the specifications must be less permissive and more deterministic. This limits the flexibility in designing specifications; in general, one want to write specifications as nondeterministically as possible. With such specifications, showing that an algorithm meets the specification involves only one-way simulation and not showing equivalence.

Erdös systems are specified in the propositional branching-time temporal logic CTL. As a formal language, CTL provides structure and guidance for users. In particular, complicated properties are often formulated as conjunctions of simple ones. On the other hand, the language structure limits the flexibility and expressiveness. e.g., specifications are restricted to finite-state systems. Verification is based on CTL model checking. The advantage of this technique is that it is fully automated, for finite-state systems.

References

[1] T. Araragi. Agent programming and its formal verification (in Japanese), technical report ai99-47, pp. 47–54. Technical report, The Institute of Electronics, Information and Communication Engineers, 1999.

[2] T. Araragi, P. Attie, I. Keidar, K. Kogure, V. Luchangco, N. Lynch, and K. Mano. On Formal Modeling of Agent Computations. Tech. Report, MIT Lab. for Comp. Sci. Url: http://theory.lcs.mit.edu/∼idish/Abstracts/agents-compare.html

[3] P.C. Attie and N.A. Lynch. A formal model for dynamic computation. Tech. report, MIT Lab. for Computer Science. To appear.

[4] E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[5] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge.* The MIT Press, Cambridge, Mass., 1995.

[6] E. Horita and K. Mano. Nepi^2: a two-level calculus for network programming based on the π-calculus. In *Proc. 3rd Asian Computing Science Conference (ASIAN'97).* Springer LNCS 1345, 1997.

[7] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sep. 1989. Also MIT/LCS/TM-373, Nov. 1988.

[8] R. Milner. *Communicating and mobile systems: the π-calculus.* Addison-Wesley, Reading, Mass., 1999.

Modeling and Programming Devices and Web Agents[⋆]

Sheila A. McIlraith

Knowledge Systems Laboratory, Department of Computer Science
Stanford University, Stanford CA 94025-9020
sam@ksl.stanford.edu

Abstract. This paper integrates research in robot programming and reasoning about action with research in model-based reasoning about physical systems to provide a capability for modeling and programming devices and web agents, which we term *model-based programming*. Model-based programs are reusable high-level programs that capture the procedural knowledge of how to accomplish a task, without specifying all the device- and web-service-specific details. Model-based programs must be instantiated in the context of a model of a specific device/web service and state of the world. The instantiated programs are simply sequences of actions, which can be executed by an appropriate agent to control the behavior of the system. The separation of control and model enables reuse of model-based programs across classes of related devices and services whose configuration changes as the result of replacement, redesign, reconfiguration or component failure. Additionally, the logical formalism underlying model-based programming enables verification of properties such as safety, program existence, and goal achievement. Our model-based programs are realized by exploiting research on the logic programming language Golog, together with research on representing actions and state constraints in the situation calculus, and modeling physical systems using state constraints.

1 Introduction

We are seeing a tremendous increase in the prevalence of physical devices with embedded digital controllers. Such devices range from smart children's toys to smart photocopiers and buildings, power distribution systems, and autonomous spacecraft. We are likewise seeing the web evolve from an information-based service provider to a network populated by programs, sensors and other devices. It is predicted that in the next decade, computers will be ubiquitous, that many device will have some sort of computer inside them, and that many business services will be agent-enabled and delivered over the web. Designing reliable software for these devices and web agents is often a complex task. In our research, we examine formal techniques to model, diagnose, test, and more recently, to program physical devices and web agents that are controlled by digital computers.

In this paper we integrate and extend research on the agent programming language Golog (e.g., [4]) and reasoning about action in the situation calculus with research in modeling physical systems and model-based reasoning about physical systems (e.g., [7,

[⋆] A preliminary version of this work was presented at the Tenth International Workshop on Principles of Diagnosis, June, 1999.

8,9]) to provide a new capability for developing device and web agent software which we term *model-based programming*[1,2]. Model-based programs are generic, reusable high-level programs that capture the generic procedural knowledge of *how to* accomplish a task for a class of devices, such as isolating valve leaks in spacecraft, without specifying all the device-specific details, such as turning off valve-54 before valve-93. Model-based programs (MBPs) are deductively instantiated in the context of a rich device-specific model of device structure, behavior and function, and state. The instantiated programs are simply sequences of actions, which are performed to realize the program.

The merits of model-based programming come from the exploitation of models of system behavior and from the separation of those models from high-level procedural knowledge about how to perform a task for *classes* of like devices. Model-based programs are written at a sufficiently high level of abstraction that they are amenable to reuse in the face of device reconfiguration, replacement, redesign or component failure. Also, they are easier to write than traditional control programs for devices, ridding the engineer/programmer of keeping track of the potentially complex details of a system design, with all its subcomponent interactions. Finally, because of the logical foundations of model-based programming, certain properties of model-based programs such as safety, program existence and goal achievement can be verified, and/or simply enforced in the generation of program instances.

In this paper, we argue that the situation calculus [6,14] and the logic programming Golog [4] together provide a natural formalism for model-based programming. To develop the models for our model-based programming paradigm, we take as our starting point: a set of state constraints in first-order logic, that can describe the structure and behavior of a physical device; and a set of actions. We appeal to a solution to the frame and ramification problems in the situation calculus in order to provide an integrated representation of our physical device and the actions that affect it. This representation scheme is the critical enabler of our model-based programming capability. It provides the representation scheme for declarative encoding of the *model* for our model-based programs. With a representation for our models in hand, we introduce the notion of a model-based program, show how to exploit Golog to specify model-based programs, and show how to generate program instances from the program and the model using deductive machinery. Finally, we show how the logical formalism underlying model-based programming enables verification of certain properties such as safety, program existence, and goal achievement. We conclude with a brief discussion of related work.

2 Model-Based Programming

Model-based programming comprises two components:

A Model which provides an integrated representation of the structure and behavior of the complex physical system being programmed, the operator or controller actions

[1] The term *model-based programming* did not originate with us (e.g., [16,18], etc.) We each use the term differently.

[2] As a result of space limitations, we restrict our discussion to devices, leaving further explicit discussion of web agents to a longer paper.

that affect it, and the state of the system. The model dictates the language for the program, and is often shared over a class of like devices.

A Program which describes the high-level procedure for performing some task, using the operator or controller actions.

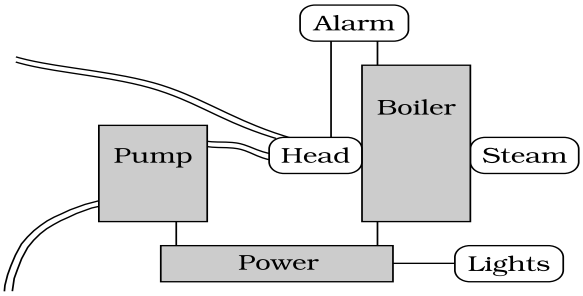
2.1 The Model

The first step towards achieving our vision of model-based programming is to define a suitable representation for our models. In this section we demonstrate that the situation calculus will provide a suitable language for declarative specification of our device models. Model-based reasoning often represents the structure and behavior of physical systems as a set of state constraints in first-order logic. The first challenge we must address is how to integrate operator or controller actions into our representation, in order to obtain an integrated representation of our system. To do so, we appeal to a solution to the frame and ramification problems proposed in [7,10], that automatically compiles a situation calculus theory of action with a a set of state constraints. We begin with a brief overview of the situation calculus.

The Situation Calculus. The situation calculus language we employ to axiomatize our domains is a sorted first-order language with equality. The sorts are of type $\mathcal{A}$ for primitive *actions*, $\mathcal{S}$ for *situations*, $\mathcal{F}$ for fluents, and $\mathcal{O}$ for everything else, including domain objects ([14]). We represent each action as a (possibly parameterized) first-class object within the language. Situations are simply sequences of actions. The evolution of the world can be viewed as a tree rooted at the distinguished initial situation S_0. The branches of the tree are determined by the possible future situations that could arise from the realization of particular sequences of actions. As such, each situation along the tree is simply a history of the sequence of actions performed to reach it. The function symbol do maps an action term and a situation term into a new situation term. For example, $do(turn\_on(Pmp, S_0))$ is the situation resulting from performing the action of turning on the pump in situation S_0. The distinguished predicate $Poss(a, s)$ denotes that an action a is possible to perform in situation s (e.g., $Poss(turn\_on(Pmp), S_0)$). Thus, $Poss$ determines the subset of the situation tree consisting of situations that are possible in the world. Finally, those properties or relations whose truth value can change from situation to situation are referred to as *fluents*. For example, the property that the pump is on in situation s could be represented by the fluent $on(Pmp, s)$. The situation calculus language we employ in this paper is restricted to primitive, determinate actions. For the present, our language does not include a representation of time or concurrency.

The Representation Scheme. Our representation scheme automatically integrates a set of state constraints, such as the ones found in a typical model-based reasoning system description, SD [2] with a situation calculus theory of action to provide a compiled representation scheme. We sketch the integration procedure in sufficient detail to be replicated. We illustrate it in terms of an example of a power plant feedwater system.

The system consists of three potentially malfunctioning components: a power supply (Pwr); a pump (Pmp); and a boiler (Blr). The power supply provides power to both the pump and the boiler. The pump fills the header with water (wtr_enter_head), which in turn provides water to the boiler, producing steam. Alternately, the header can be filled manually (Man_Fill). To make the example more interesting, we take liberty with the functioning of the actual system and assume that once water is entering the header, a siphon is created. Water will only stop entering the header when the siphon is stopped. The system also contains lights and an alarm, and it contains people. The plant is occupied at all times unless it is explicitly evacuated. Finally we have stipulated certain components of the plant as vital. Such components should not be turned off in the event of an emergency.



Power Plant Feedwater System

This system is typically axiomatized in terms of a set of **state constraints**. The following is a representative subset.[3] All formulae are universally quantified with maximum scope, unless otherwise noted.

$$\neg AB(Pwr) \wedge \neg AB(Pmp) \wedge on(Pmp) \supset wtr\_enters\_head$$
$$on(Man\_Fill) \supset wtr\_enters\_head$$
$$\neg wtr\_enters\_head \wedge on(Blr) \supset on(Alarm)$$
$$AB(Blr) \supset on(Alarm)$$
$$\ldots$$
$$\neg(on(Pmp) \wedge on(Man\_Fill))$$
$$Pwr \neq Pmp \neq Blr \neq Aux\_Pwr \neq Alarm \neq Man\_Fill$$

We also have a situation calculus action theory. One component of our theory of action is a set of **effect axioms** that describe the effects on our power plant of actions performed by the system, a human or nature. The effect axioms take the following form:

$$Poss(a, s) \wedge \textsf{conditions} \supset fluent(\boldsymbol{x}, do(a, s)).$$

Effect axioms state that if $Poss(a, s)$, i.e. it is possible to perform action a in situation s, and some conditions are true, then *fluent* will be true in the situation resulting from doing action a in situation s, i.e. the situation $do(a, s)$. The following are typical effect axioms.

$$Poss(a, s) \wedge a = turn\_on(Pmp) \supset on(Pmp, do(a, s))$$
$$Poss(a, s) \wedge a = blr\_blow \supset AB(Blr, do(a, s))$$

[3] Note that for simplicity, this particular set of state constraints violates the no-function-in-structure philosophy. This characteristic is not in any way essential to our representation.

In addition to effect axioms our theory also has a set of **necessary conditions for actions** which are of the following general form:

$$Poss(a, s) \supset \textsf{nec\_conditions}$$

These axioms say that if it is possible to perform action a in situation s then certain conditions (so-called nec_conditions) must hold in that situation. The following are typical necessary conditions for actions.

$$Poss(blr\_blow, s) \supset on(Blr, s)$$
$$Poss(blr\_fix, s) \supset \neg on(Blr, s)$$

We now have axioms describing the constraints on the system state, and also axioms describing the actions that affect system state. Unfortunately, these axioms collectively yield unintended interpretations. That is, there are unintended (semantic) models of this theory. This happens because there are several assumptions that we hold about the theory that have not been made explicit. In particular,

Completeness Assumption: we assume that the axiomatizer has done his/her job properly and that the state constraints, effect axioms and necessary conditions for actions capture all the elements that can affect our system.

Causal Structure: we assume a particular causal structure that lets us interpret how the actions interact with our state constraints, i.e. how effects are propagated through the system, and what state constraints preclude an action from being performed. The causal structure must be acyclic.

We make these assumptions explicit and compile our assumptions, state constraints and theory of action into a final model-based representation. The compilation process is semantically justified and fully described in [10]. The resulting example axiomatization is provided below. We will refer to this collection of axioms as a **situation calculus domain axiomatization** and together with foundational axioms of the situation calculus, Σ [14] they form a situation calculus theory, $\mathcal{D}$. • successor state axioms, $\mathcal{D}_{SS}$,

- action precondition axioms, $\mathcal{D}_{ap}$,
- axioms describing the initial situation, $\mathcal{D}_{S_0}$,
- unique names for actions, $\mathcal{D}_{una}$,
- domain closure axioms for actions, $\mathcal{D}_{dca}$.

The first element of the domain axiomatization after compilation is the set of **successor state axioms**, compiled form the effect axioms and state constraints under the assumptions above. Successor state axioms are of the following general form.

$$\begin{aligned} Poss(a, s) \supset [fluent(do(a, s)) \equiv\ & \textsf{an action made it true} \\ \vee\ & \textsf{a state constraint made it true} \\ \vee\ & \textsf{it was already true} \\ & \quad \wedge\ \textsf{neither an action nor a state constraint made it false}] \end{aligned}$$

I.e., if it is possible to perform action a in situation s, then $fluent$ will be true in the resulting situation if and only if an action made it true, a state constraint made it true, or it was already true and neither an action nor a state constraint made it false.

The set of *intermediate* successor state axioms for our example:

$$Poss(a,s) \supset [on(Pmp, do(a,s)) \equiv a = turn\_on(Pmp) \\ \vee (on(Pmp,s) \wedge a \neq turn\_off(Pmp))] \quad (1)$$

$$Poss(a,s) \supset [on(Aux\_Pwr, do(a,s)) \equiv a = turn\_on(Aux\_Pwr) \\ \vee (on(Aux\_Pwr,s) \wedge a \neq turn\_off(Aux\_Pwr))] \quad (2)$$

$$Poss(a,s) \supset [on(Blr, do(a,s)) \equiv a = turn\_on(Blr) \\ \vee (on(Blr,s) \wedge a \neq turn\_off(Blr))] \quad (3)$$

$$Poss(a,s) \supset [on(Alarm, do(a,s)) \equiv a = turn\_on(Alarm) \vee AB(Blr, (do(a,s)) \\ \vee (\neg wtr\_enter\_head(do(a,s)) \wedge on(Blr, do(a,s))) \\ \vee (on(Alarm,s) \wedge a \neq turn\_off(Alarm)] \quad (4)$$

$$Poss(a,s) \supset [AB(Blr, do(a,s)) \equiv a = blr\_blow \vee (AB(Blr,s) \wedge a \neq blr\_fix)] \quad (5)$$

$$Poss(a,s) \supset [AB(Pwr, do(a,s)) \equiv a = pwr\_failure \\ \vee (AB(Pwr,s) \wedge a \neq turn\_on(Aux\_Pwr) \\ \wedge\ a \neq pwr\_fix)] \quad (6)$$

$$Poss(a,s) \supset [AB(Pmp, do(a,s)) \equiv a = pmp\_burn\_out \\ \vee (AB(Pmp,s) \wedge a \neq pmp\_fix)] \quad (7)$$

$$Poss(a,s) \supset [on(Man\_Fill, do(a,s)) \equiv a = turn\_on(Man\_Fill) \\ \vee (on(Man\_Fill,s) \wedge a \neq turn\_off(Man\_Fill))] \quad (8)$$

$$Poss(a,s) \supset [wtr\_enter\_head(do(a,s)) \equiv on(Man\_Fill, do(a,s)) \\ \vee (\neg AB(Pwr, do(a,s)) \wedge \neg AB(Pmp, do(a,s)) \\ \wedge\ on(Pmp, do(a,s))) \\ \vee wtr\_enter\_head(s) \wedge a \neq stop\_siphon] \quad (9)$$

$$Poss(a,s) \supset [lights\_out(do(a,s)) \equiv AB(Pwr, do(a,s)) \\ \wedge \neg on(Aux\_Pwr, do(a,s))] \quad (10)$$

$$Poss(a,s) \supset [steam(do(a,s)) \equiv (wtr\_enter\_head(do(a,s)) \wedge \neg AB(Pwr, do(a,s)) \\ \wedge\ \neg AB(Blr, do(a,s)) \wedge on(Blr, do(a,s)))] \quad (11)$$

$$Poss(a,s) \supset [occupied(do(a,s)) \equiv (occupied(s) \wedge a \neq evacuate)] \quad (12)$$

These are *intermediate* successor state axioms because they can be further compiled by substituting other intermediate successor state axioms for fluents relativized to situation

$do(a,s)$ on the righthand side of the equivalence connective. For example, Axiom (5) can be substituted into Axiom (4).

Additionally, there is a set of **action precondition axioms** that capture the necessary and sufficient conditions for actions. They are a compilation of the necessary conditions for actions and the state constraints under the assumptions above. They are of the form:

$$Poss(a,s) \equiv \text{nec\_conditions} \wedge \text{ implicit conditions from state constraints}$$

For example,

$$Poss(blr\_blow, s) \equiv \neg wtr\_enter\_head(s) \wedge on(Blr, s) \tag{13}$$
$$Poss(pmp\_burn\_out, s) \equiv on(Pmp, s) \tag{14}$$
$$Poss(blr\_fix, s) \equiv \neg on(Blr, s) \tag{15}$$
$$Poss(turn\_off(Alarm), s) \equiv (wtr\_enter\_head(s) \vee \neg on(Blr, s)) \wedge \neg AB(Blr, s) \tag{16}$$
$$Poss(turn\_on(Man\_Fill), s) \equiv \neg on(Alarm, s) \wedge \neg on(Pmp, s) \tag{17}$$
$$Poss(turn\_on(Pmp), s) \equiv \neg on(Man\_Fill, s) \tag{18}$$
$$Poss(turn\_on(Alarm), s) \equiv true \tag{19}$$
$$\ldots$$

Our axiomatization will specify what is known of the **initial situation** of the world. This will include the truth value of some fluents in the initial situation S_0. E.g.,

$$vital(Pwr) \wedge vital(Aux\_Pwr) \wedge vital(Alarm) \tag{20}$$
$$\neg vital(Pmp) \wedge \neg vital(Blr) \wedge \neg vital(Man\_Fill) \tag{21}$$
$$\neg on(Pmp, S_0) \wedge \neg on(Blr, S_0) \wedge \neg on(Man\_Fill, S_0) \tag{22}$$
$$\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge \neg AB(Blr, S_0) \tag{23}$$
$$\neg on(Aux\_Pwr, S_0) \wedge on(Pwr, S_0) \wedge occupied(S_0) \tag{24}$$

It will also include the state constraints relativized to the initial situation. E.g.,

$$\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge on(Pmp, S_0) \supset wtr\_enter\_head(S_0) \tag{25}$$
$$on(Man\_Fill, S_0) \supset wtr\_enter\_head(S_0) \tag{26}$$
$$wtr\_enter\_head(S_0) \wedge \neg AB(Pwr, S_0) \wedge \neg AB(Blr, S_0) \wedge on(Blr, S_0) \supset steam(S_0) \tag{27}$$
$$\neg wtr\_enter\_head(S_0) \wedge on(Blr, S_0) \supset on(Alarm, S_0) \tag{28}$$
$$AB(Blr, S_0) \supset on(Alarm, S_0) \tag{29}$$
$$\ldots$$

We have demonstrated that the situation calculus provides a suitable representation for the model-based programming models.

Definition 1 (Model). *A model-based programming model, M is a situation calculus domain axiomatization on the situation calculus language $\mathcal{L}$.*

We henceforth refer to the model of our power plant feedwater example as M_{SD}.

2.2 The Program

With the critical model representation in hand, we must now find a suitable representation for our model-based programs. Further, we must find a suitable mechanism for instantiating our model-based program with respect to our models. We argue that the logic programming language, Golog and theorem proving provide a natural formalism for this task. In the subsection to follow, we introduce the Golog logic programming language and its exploitation for model-based programming.

Golog. Golog is a high-level logic programming language developed at the University of Toronto (e.g., [4]). Its primary use is for robot programming and to support high-level robot task planning (e.g., [1]), but it has also been used for agent-based programming (e.g., meeting scheduling). Golog provides a set of extralogical *constructs* for assembling *primitive actions* defined in the situation calculus (e.g., $turn\_on(Blr)$ or $stop\_siphon$ in our power plant example) into *macros* that can be viewed as complex actions, and that assemble into a program.

In the context of our model-based representation, we can define a set of macros that is relevant to our domain or to a family of systems in our domain. The instruction set for these macros, the primitive actions, are simply the domain-specific primitive actions of our model-based representation. Hence, the macros or complex actions simply reduce to first-order (and occasionally second-order) formulae in our situation calculus language. The following are examples of Golog statements.

```
if AB($Pmp$) then PMP_FIX endIf

while $(\exists x)$.ON($x$) do TURN_OFF($x$) endWhile

proc PREVENTDANGER
    if OCCUPIED then EVACUATE endIf
endProc
```

We leave detailed discussion of Golog to [4,14] and simply describe the constructs for the Golog language. Let δ_1 and δ_2 be complex action expressions and let ϕ and a be so-called **pseudo fluents/actions**, respectively, i.e., a fluent/action in the language of the situation calculus with all its situation arguments suppressed.

primitive action	a
test of truth	$\phi?$
sequence	$(\delta_1; \delta_2)$
nondeterministic choice between actions	$(\delta_1 \mid \delta_2)$
nondeterministic choice of arguments	$\pi x.\delta$
nondeterministic iteration	δ^*
conditional	**if** ϕ **then** δ_1 **else** δ_2 **endIf**
loop	**while** ϕ **do** δ **endWhile**
procedure	**proc** $P(\boldsymbol{v})$ δ **endProc**

A Golog **program** can in turn be comprised of a combination of procedures.

Golog also defines the abbreviation $Do(\delta, s, s^{'})$. It says that $Do(\delta, s, s^{'})$ holds whenever $s^{'}$ is a terminating situation following the execution of complex action δ, starting in situation s. Under Do, each of the programming constructs listed above is simply a macro, equivalent to a situation calculus formula.

Do is defined for each complex action construct. Three are defined below.

$$Do(a, s, s^{'}) \doteq Poss(a[s], s) \wedge s^{'} = do(a[s], s)^{4}$$
$$Do([\delta_1; \delta_2], s, s^{'}) \doteq (\exists s^{*}).(Do(\delta_1, s, s^{*}) \wedge Do(\delta_2, s^{*}, s^{'}))$$
$$Do((\pi x)\delta(x), s, s^{'}) \doteq (\exists x).Do(\delta(x), s, s^{'})$$

Definitions of the rest of the complex actions can be found in [4] but their meaning should be apparent from the examples below. Before returning to our example, we define what we mean by a model-based program.

Definition 2 (Model-Based Program, δ for model M). *Given a model M in situation calculus language $\mathcal{L}$, δ is a model-based program for model M iff δ is a Golog program that only mentions pseudo actions and pseudo fluents drawn from $\mathcal{L}$.*

We begin by defining a rather simple looking procedure to illustrate the constructs in our language and to illustrate the range of procedures Golog can instantiate with respect to the example model, M_{SD}.

```
proc SHUTDOWN
    $\forall(x)[\text{VITAL}(x) \vee \text{OFF}(x)]? \mid$
        $(\pi x)[[\text{ON}(x) \wedge \neg\, \text{VITAL}(x)]?;\ \text{TURNOFF}(x)];\text{SHUTDOWN}$
endProc
```

The procedure SHUTDOWN directs the agent to turn off everything that isn't vital. If it is not the case that either everything is off or else it is vital, then pick a random thing that is on and that is not vital, turn it off and repeat the procedure until everything is either off or else it is vital.

From the simple procedures defined above, we can define the following model-based program that dictates a procedure for addressing an abnormal boiler.

```
if AB($Blr$) then
    PREVENTDANGER; SHUTDOWN; BLR_FIX; RESTART[5]
end if                                                    (30)
```

This program on its own is very simple and seems uninteresting since it exploits little domain knowledge and thus doesn't capture many of the idiosyncrasies of the system. Instead, it illustrates the beauty of model-based programming. By using nondeterministic choice, the program need not stipulate which component to turn off first, but if there is a physical requirement to turn one component off before another, then it will be

[4] Notation: $a[s]$ denotes the restoration of the situation arguments to any functional fluents mentioned by the action term a.

[5] Procedure not defined here.

dictated in the model, M of the specific system, and when the model-based program is instantiated, M will ensure that the instantiation of the program enforces this ordering. This use of nondeterminism and exploitation of the model makes the program reusable for multiple different devices without the need to rewrite the program. It also saves the engineer/programmer from being mired in the details of the physical constraints of a potentially complex specific system.

It is important to observe that model-based programs are not programs in the conventional sense. While they have the complex structure of programs, including loops, if-then-else statements etc., they differ in that they are not necessarily deterministic. As such they run the gamut from playing the role of a procedurally specified plan sketch that helps to constrain the search space required in planning, to the other extreme where the model-based program provides a deterministic sequence of actions, much in the way a traditional program might. Unfortunately, planning is hard, particularly in cases where we have incomplete knowledge. Computationally, in the worst-case, a model-based program will further constrain the search space, helping the search engines hone in on a suitable sequence of actions to achieve the objective of the program. In the best place, it will dictate a unique sequence of actions.

Indeed, what makes Golog ideal for model-based programming, is how Golog programs are instantiated with respect to a model.

Definition 3 (Model-Based Program Instance, $\boldsymbol{A}$). *$\boldsymbol{A}$ is a model-based program instance of model M and model-based program δ iff $\boldsymbol{A}$ is a sequence of actions $[a_1, \ldots, a_m]$ such that*

$$M \models Do(\delta, S_0, do([a1, \ldots, a_m], S_0))^6.$$

Recall that the program itself is simply a macro for one or more situation calculus formulae. Hence, generation of a program instance can be achieved by theorem proving, in particular, by trying to prove $\exists s^{'}.Do(\delta, S_0, s^{'})$ from model M. The sequence of actions, $[a1, \ldots, a_m]$ constituting the program instance can be extracted from the binding for $s^{'}$ in the proof. We can see that in this context, the instantiation of a model-based program is related to deductive plan synthesis [3].

Returning to our example, instantiating the model-based program (30) with respect to our example model M_{SD}, which includes some constraints on the initial situation S_0 as defined in Axioms (20)–(24), terminates at the situation $do(evacuate, S_0)$. Consequently, the model-based program instance is composed of the single action $evacuate$. (All the other components of the system are off in the initial situation.) If the initial situation were changed so that all components that could be on at the same time were on, the proof of the program might return the terminating situation

$$do(turn\_off(Pmp), do(turn\_off(Blr), do(turn\_off(Alarm), do(evacuate, S_0))))$$

thus yielding the model-based program instance

$$evacuate;\ turn\_off(Alarm);\ turn\_off(Blr);\ turn\_off(Pmp).$$

To illustrate the power of Golog as a model-based programming language, imagine that our system is more complex than the one described by M_{SD}, that the pump must

6 Notation: $do([a1, \ldots, a_m], S_0)$ abbreviates $do(a_m, (do(a_{m-1}, \ldots, (do(a_1, S_0)))))$.

be turned off after the boiler, and that before the boiler is turned off that there are valves that must be turned off. If this knowledge is contained in the model M_{SD2}, then this same simple model-based program, (30) is still applicable, but its instantiation will be different. In particular, to instantiate this model-based program, the theorem prover will pick a random nonvital component to turn off, but the preconditions to turn off that component may not be true, if so it will pick another, and another until it finally finds the correct sequence of actions that constitutes a proof, and hence a legal action sequence.

In this instance, an alternative to shutDown would be to exploit the knowledge of an expert familiar with the device, and to write a device-specific shutdown procedure, along the lines of the following, that captures at least some of this device-specific procedural knowledge.

```
**proc** newShutDown
    shutValves; turnOff($Blr$); turnOff($Pmp$); turnOff($Alarm$)
**endProc**

**proc** shutValves
    $\forall(x)$[valve$(x)$ $\supset$ off$(x)$]? |
        $(\pi x)$[[valve$(x)$ $\wedge$ $\neg$ on$(x)$]?; turnOff$(x)$]; shutValves
**endProc**
```

Indeed, in this particular example, writing such a program is viable, and newShutDown captures the expertise of the expert and in so doing, makes the the model-based instantiation process more efficient. Nevertheless, with a complex physical system comprised of hundreds of complex interacting components, correct sequencing of a shutdown procedure may be better left to a theorem prover following the complex constraints dictated in the model, rather than expecting a control engineer to recall all the complex interdependencies of the system.

This last example serves to illustrate that model-based programs can reside along a continuum from being underconstrained articulations of the goal of a task, to being a predetermined sequence of actions for achieving that goal. shutDown is situated closer to the goal end of the spectrum, whereas newShutDown is closer towards a predetermined sequence of actions.

3 Proving Properties of Programs

It is often desirable to be able to enforce and/or prove certain formal properties of programs. In our model-based programming paradigm, we may wish to verify properties of a model-based program we have written or of a program instance we have generated. We may also wish to experiment with the behavior of our model-based program by modifying aspects of our model M and seeing what effect it has on program properties. A special case of this, is modifying the initial situation S_0. Finally, rather than verifying properties, we may wish to actually generate program instances which enforce certain properties. Since our model-based programs are simply macros for logical expressions, our programming paradigm immediately lends itself to this task.

Recall that model-based programs are generally written as generic procedures for *classes* of devices. Hence, an important first property to prove is that a program instance actually *exists* for a particular model-based program and specific device model. This proposition also shows that the program terminates [4].

Proposition 1 (Program Instance Existence). *A program instance exists for model-based program δ and model M iff*

$$M \models \exists s.Do(\delta, S_0, s).$$

Another interesting property is safety. Engineers who write control procedures often wish to verify that the trajectories generated by their control procedures do not pass through unsafe states, i.e., states where some safety property P does not hold.

Proposition 2 (Program Instance Safety). *Let $P(s)$ be a first-order formula representing the safety property. A program instance, $\boldsymbol{A} = [a_1, \ldots, a_m]$ of model-based program δ and model M enforces safety property $P(s)$ iff*

$$M \models Do(\delta, S_0, do(\boldsymbol{A}, S_0)) \supset P(\boldsymbol{A}, S_0).$$[7]

By a simple variation on the above proposition, we can prove several stronger safety properties. For example, we can prove that a model-based program enforces the safety property for every potential program instance.

Proposition 3 (Program Safety). *Let $P(s)$ be a first-order formula representing the safety property. A model-based program, δ and model M enforce safety property $P(s)$ iff there is no situation s such that*

$$M \models \exists s.Do(\delta, S_0, s) \supset \neg P(\boldsymbol{\alpha}, S_0),$$

where for each situation variable $s = do([\alpha_1, \ldots, \alpha_n], S_0)$, $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]$.

A final property we wish to examine is goal achievement. Since our model-based programs are designed with some task in mind, we may wish to prove that when the program has terminated execution, it will have achieved the desired goal.

Proposition 4 (Program Instance Goal Achievement). *Let $G(s)$ be a first-order formula representing the goal of model-based program δ. A program instance, $\boldsymbol{A} = [a_1, \ldots, a_m]$ of model-based program, δ and model M achieves the goal $G(s)$ iff*

$$M \models Do(\delta, S_0, do(\boldsymbol{A}, S_0)) \supset G(do(\boldsymbol{A}, S_0)).$$

There are many variants on these and other propositions, regarding properties of programs. For example, up until now, we have assumed that we have a fixed initial situation S_0, whose state is captured in our model, M. We can strengthen many of the above propositions by rejecting this assumption and proving Propositions 1, 3 for

[7] Notation: $do(\boldsymbol{A}, S_0)$ is an abbreviation for $do(a_m, (do(a_{m-1}, \ldots, (do(a_1, S_0)))))$. $P(\boldsymbol{A}, S_0)$ is an abbreviation for $P(S_0) \wedge P(do(a_1, S_0)) \wedge \ldots \wedge P(do(\boldsymbol{A}, S_0))$.

any initial situation. This can be done by replacing S_0 by initial situation variable s_0 and by quantifying, not only over s, but universally quantifying over s_0. Clearly, many programs will not enable the proof of properties for all initial situations, but the associated propositions still hold.

Finally, we can exploit automated reasoning techniques to prove some of these properties. In particular, for procedure and while-loop free programs, we can exploit regression [17] followed by theorem proving in the initial situation, as we do to prove other queries. We leave discussion of this topic to another paper.

4 Related Work

The work presented in this paper is related to several different research areas. In particular, this research is related in spirit only to work on plan sketches such as [11]. In contrast, plan sketches are instantiated through hierarchical substitution. Further, plan sketches generally don't exploit the procedural programming language constructs found in our model-based programming language. Model-based programming is also related to various types of program synthesis and model-based software reuse (e.g., [5,15,16]) and to model-based generation of decision trees (e.g., [13]). A subtle distinction is that whereas deductive program synthesis uses deductive machinery to synthesize a program from a specification, model-based programming *starts* with a program, and uses models and deductive machinery to simply fill in some details. Finally, model-based programming is related to planning and in particular to deductive plan synthesis (e.g., [3]). A Golog program, despite its if-then-else's and while loops, can be viewed as extra formulae that are added to the domain axiomatization, including formulae that define the terminating (or goal) situation. In so doing, these formulae reduce the search space required to search for or plan a sequence of actions.

Needless to say, model-based programming is intimately related to cognitive robotics, agent-based programming, and robot programming, particularly in Golog. This work drew heavily from the research on Golog. A major distinction in our work has been the challenge of dealing with large numbers of state constraints inherent to the representation of complex physical systems, and the desire to prove certain properties of our programs. In the first regard, our work is related to ongoing work at NASA on immobots [18], and in particular to research charged with developing a model-based executive.

Finally, this work is related to controller synthesis and controller programming from the engineering community. Comments on the distinction between model-based programming and program synthesis also hold for controller synthesis. With respect to controller programming, typical controller programming languages do not separate control from models. Hence, programs are system specific and not model based. As a consequence they are harder to write, much more brittle, and are not generally amenable to reuse across classes of devices.

5 Summary and Discussion

The main contribution of this paper was to propose and provide a capability for programming devices and web agents. Specifically: we envisaged the concept of model-based programming; proposed a representation and compilation procedure to create suitable models of physical systems in the situation calculus; proposed and demonstrated the effectiveness of Golog for expressing model-based programs themselves, and theorem proving as a model-based program instantiation mechanism. We also provided a set of propositions that characterized interesting properties of programs that could be verified or enforced within our model-based programming framework via regression and theorem proving.

The merits of model-based programming come from the exploitation of models of system behavior and from the separation of those models from high-level procedural knowledge about how to perform a task. Model-based programs are written at a sufficiently high level of abstraction that they are very amenable to reuse over classes of devices. Also, they are easier to write than traditional control programs, ridding the engineer/programmer of keeping track of the potentially complex details of a system design, with all its subcomponent interactions. Further, because of the logical foundations of model-based programming, important properties of model-based programs such as safety, program existence and goal achievement can be verified, and/or simply enforced in the generation of program instances.

There are several weaknesses to our approach at this time, The first is inherent in Golog – not all complex actions comprising our Golog programming language are first-order definable. Hence, in its general form, our model-based programming language is second order. However, as observed by [4] and experienced by the authors, first order is adequate for most purposes. The second problem is that the Prolog implementation of Golog relies on a closed-world assumption (CWA) which has suited our purposes, but is not a valid assumption in the general case. Finally, not all physical system behavior can be expressed as logical state constraints. This can be addressed by extending our model representation language to include ODE's (e.g., [12]).

Acknowledgements. We gratefully acknowledge the Cognitive Robotics Group at the University of Toronto for work on the development and implementation of Golog, and related research in the situation calculus. We also acknowledge NASA Grant NAG2-1337 and DARPA Grant N66001-97-C-8554-P00004 for partial funding of this work.

References

1. W. Burgard, A.B. Cremers, D. Fox, D. Haehnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 11–18, 1998.
2. J. de Kleer, A.K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence,* 56(2–3):197–222, 1992.
3. C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence* 4, pages 183–205. American Elsevier, New York, 1969.
4. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming,* 31:59–84, 1997.
5. Z. Manna and R. Waldinger. How to Clear a Block: A Theory of Plans. *Journal of Automated Reasoning,* 3:343–377, 1987.

6. J. McCarthy. Programs with common sense. In Marvin Minsky, editor, *Semantic Information Processing,* chapter 7, pages 403–418. The MIT Press, 1968.
7. S. McIlraith. Representing actions and state constraints in model-based diagnosis. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97),* pages 43–49, 1997.
8. S. McIlraith. *Towards a Formal Account of Diagnostic Problem Solving.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1997.
9. S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98),* pages 167–177, 1998.
10. S. McIlraith. A closed-form solution to the ramification problem (sometimes). *Artificial Intelligence,* 116(1–2):87–121, 2000.
11. K. Myers. Abductive completion of plan sketches. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97),* pages 687–693, 1997.
12. J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1994.
13. C. Price, M. Wilson, J. Timmis, and C.Cain. Generating fault trees from fmea. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis,* pages 183–190, 1996.
14. R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* 2000. In preparation. Draft available at `http://www.cs.toronto.edu/~cogrobo/`.
15. D. Smith and C. Green. Towards Practical Application of Software Synthesis. In *Proceedings of FMSP'96, the First Workshop on Formal Methods in Software Practice,* pages 31–39, San Diego, CA, January 1996.
16. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction,* 1994.
17. R. Waldinger. Achieving several goals simultaneously. In E. Elcock and D. Michie, editors, *Machine Intelligence 8,* pages 94–136. Ellis Horwood, Edinburgh, Scotland, 1977.
18. B. Williams and P. Nayak. Immobile robotics: AI in the new millenium. *AI Magazine,* pages 16–35, 1996.

A Programming Logic for Part of the Agent Language 3APL

Koen V. Hindriks[1], Frank S. de Boer[1], Wiebe van der Hoek[1,2,3], and John-Jules C. Meyer[1]

[1] Institute of Information and Computing Sciences,
Utrecht University, The Netherlands
{koenh,frankb,wiebe,jj}@cs.uu.nl
[2] Department of Philosophy
Utrecht University, The Netherlands
Wiebe.vanderHoek@phil.uu.nl
[3] Department of Computer Science
The University of Liverpool
United Kingdom

Abstract. 3APL is an agent programming language based on the concept of an *intelligent agent*. An intelligent agent is a computational entity with a mental state consisting of its *beliefs* and *goals*. The *operational semantics* of the language 3APL is specified by a formal semantics in terms of a socalled *transition system*. An operational semantics allows operational reasoning about agents, but does not allow for a *compositional* style of reasoning based on the *structure* of the agent itself. For this purpose, in this paper we construct a *denotational semantics* which corresponds to the operational semantics and provides the basis for a semantics of a *programming logic* for (part of) 3APL. The *programming logic* is a variant of a modal logic with operators for reasoning about the actions and the beliefs of an agent. Our results clarify the relation between more practical approaches to agents, represented by agent programming languages, and more theoretical work on agents, represented by socalled agent logics.

1 Introduction

One of the main challenges in the field of intelligent agent programming is to provide a satisfactory account of the relation between formal, logical specifications of intelligent agents and the agents in agent programming languages. Our work on the agent programming language 3APL (pronounced “triple-a-p-l”; cf. [4, 5]) has been an attempt to bridge the gap between the two by introducing a well defined programming language to build and implement agents.

In [4] we discuss the agent programming language 3APL (pronounced “triple-a-p-l”) in detail. The programming language 3APL is a combination of imperative programming and logic programming. From imperative programming the language inherits the full range of regular programming constructs. These constructs are used for determining the course of action of an agent. From logic programming, the language inherits the proof as computation model as a basic means of computation for querying the beliefs of an agent. 3APL is a language closely related to a number of other agent programming languages like AGENT-0 ([12]), AgentSpeak(L) ([11]), and GOLOG ([9]) as we have shown elsewhere [3, 2, 7]. There are at least two perspectives possible on what the programming language 3APL is. First, it is an *agent programming language* which supports the construction of programs by viewing them as intelligent agents. Secondly, it can be viewed upon as a language for *knowledge-based* or *database programming.*

For the specification and verification of agent systems we need a *programming logic* that enables us to reason about agents written in an agent language like 3APL. So far, no satisfying account of the relationship between agent *programming languages* and agent *logics* has been given. In this paper, we introduce a programming logic for (part of) the agent programming language 3APL to bridge the gap between agent programming frameworks and agent logics. We present a general framework for studying a *family* of *agent programming languages*, or knowledge update languages, and their related *proof logics.* The framework is general in the sense that we study abstract actions and not just any particular set of concrete actions. Concrete actions, like an insert action $\mathsf{ins}(\varphi)$ or delete action $\mathsf{del}(\varphi)$ for a proposition φ, can then be plugged in into the general framework, and a programming logic for that particular language is then obtained in a straightforward way.

In previous papers ([5, 4]) presenting the language 3APL, an *operational semantics* was provided by means of a Plotkin-style transition system ([10]). The main contributions of this paper are that we provide a *denotational semantics* and a *programming logic* for the language. The programming logic is a variant of a modal logic for reasoning about 'regular' 3APL agents (which means that the agents are build by means of the regular programming constructs from imperative programming like sequential composition, etc.). The logic provides operators for reasoning about the actions as well as the beliefs of an agent. Moreover, we prove an equivalence result which shows that the operational semantics and denotational semantics are equivalent for knowledge bases, given certain relations between the operational and denotational semantics of basic actions. The semantics of the programming logic is based on this *denotational semantics* for the programming language. The equivalence result both shows that the language 3APL is mathematically well founded, and relates the programming logic to 3APL agents. The formal equivalence result thereby makes sure that properties proven in the logic are properties of 3APL agents. The particular properties we take into account in this paper are *partial correctness properties.*

2 The Agent Programming Language 3APL

Most agent researchers agree that agents have a *complex mental state* which is made up of an informational component like *beliefs* and a motivational component like *intentions* or *goals.* Agents are supposed to be pro-active as well as reactive and may have reflective capabilities to modify their plans and goals.

Accordingly, the agent programming language 3APL has operators for manipulating the beliefs of an agent, and so-called practical reasoning rules which enable an agent to plan for new or revise old intentions. Here, intentions are taken to be plan or program like structures. In this section, we review part of the agent programming language and introduce its formal operational semantics. The focus is on the beliefs and goals of the agent and we do not discuss the practical reasoning rules of 3APL (cf. [4]). In the next section we then show how to design a logical semantics which can serve as the basis for a semantics for a programming logic for 3APL.

To represent their beliefs, agents need some knowledge representation language. In principle, this can be any language, but here we will use a first order language as the knowledge representation language used by 3APL agents. Because it is important for presenting the programming logic later on, we explicitly define this language. As is usual, terms are defined from a non-empty set of constants Cons, function symbols Func, and countably infinite variables Var. The set of all terms is denoted by Term.

Definition 1. *(knowledge representation language)*
The knowledge representation language $\mathcal{L}_0$ is then inductively defined by:

1. $p(t_1, \ldots, t_n) \in \mathcal{L}_0$ for predicates $p \in \mathsf{Pred}$ and terms $t_1, \ldots, t_n \in \mathsf{Term}$,
2. if $\phi_1, \phi_2 \in \mathcal{L}_0$, then $\neg\phi_1, \phi_1 \wedge \phi_2 \in \mathcal{L}_0$, and
3. if $x \in \mathsf{Var}$, $\phi \in \mathcal{L}_0$, then $\forall\, x(\phi) \in \mathcal{L}_0$.

By convention, we use ϕ (possibly subscripted) for formulas from $\mathcal{L}_0$. $\mathcal{L}_0$ is used to represent the agent's beliefs.

Definition 2. A *belief base* σ of an agent is any set of *sentences* from $\mathcal{L}_0$.

As mentioned in the introduction, the proof as computation model of logic programming is used to enable an agent to query its beliefs. For this reason, substitutions play an important role in the semantics of the language. However, we only use simple *grounding substitutions* and therefore our definition of substitution differs somewhat from more standard definitions.

Definition 3. *(substitution)*

1. A *substitution* θ is a set of bindings $x = c$, where $x \in \mathsf{Var}$ and $c \in \mathsf{Cons}$,
2. The domain of a substitution is denoted by $dom(\theta)$, and defined as: $dom(\theta) = \{x \mid \exists\, c(x = c \in \theta)\}$,
3. Given an expression e and a substitution θ, the expression $e\theta$ is the expression obtained by simultaneously substituting c_i for x_i for all $x_i = c_i \in \theta$,
4. The composition of two substitutions θ, γ is defined by:
   $\theta\gamma = \theta \cup \{x = t \mid x = t \in \gamma, x \notin dom(\theta)\}$.

We introduce some notation. $\varnothing$ denotes the empty substitution. The set of all substitutions is denoted by Subs.

The second basic notion associated with agents is that of a *goal.* A goal in 3APL is similar to an imperative program. Assignment statements that are the basic actions from imperative programming, however, are replaced in 3APL with actions for manipulating high-level information. Goals are built from these basic actions $\mathsf{a}(t)$, and tests $\phi?$ where ϕ is a formula from $\mathcal{L}_0$, possibly containing free variables. Basic actions define the *agent capabilities* and the execution of such a capability results in changes to the mental state of the agent. Basic actions thus are *mental state transformers*. More concretely, a basic action could range from sensory actions performed by a robot to database updates. Tests do not change the beliefs of the agent, but can be used by the agent to introspect its beliefs and to compute values or *bindings* for free variables in the test as in logic programming. More complex goals can be constructed by composing two goals with one of two programming constructs, sequential composition ; and nondeterministic choice +. We refer the reader to [4, 8] for a discussion of the other constructs like practical reasoning rules, procedure abstractions and concurrency and communication in 3APL.

Definition 4. *(goals)*
Let Bact be the set of all basic actions. Then the set of goals Goal is inductively defined as:

1. $\mathsf{Bact} \subseteq \mathsf{Goal}$,
2. $\phi? \in \mathsf{Goal}$, for $\phi \in \mathcal{L}_0$,
3. if π_1, π_2, then $\pi_1;\ \pi_2, \pi_1 + \pi_2 \in \mathsf{Goal}$,

Operational Semantics. The operational semantics of the programming language is provided by a socalled transition semantics which defines the possible computation steps that an agent can perform given its current mental state. The symbol $\longrightarrow$ is used to denote a computation step. After performing a computation step both the beliefs and the goals of the agent may have changed. The computation step relation $\longrightarrow$ is a relation on mental states and is inductively defined by means of a *transition system*. A transition system consists of a set of derivation rules of the form $\dfrac{premise(s)}{conclusion}$. A transition rule allows to derive new possible computation steps from given computation steps listed in the premises. A special symbol E is used below to denote *successful termination*, and $E;\ \pi$ is identified with π.

The semantics of the basic capabilities of the agent are not fixed except for their type. Basic actions provide an agent with update capabilities upon its beliefs. Because the specifics of these basic actions are not fixed by 3APL but are considered plug in features, we assume a socalled *transition function* $\mathcal{T}$ of type : $\mathsf{Bact} \times \wp(\mathcal{L}_0) \rightharpoonup \wp(\mathcal{L}_0)$ that specifies what type of update is associated with a basic action $\mathsf{a}(t) \in \mathsf{Bact}$. The execution of a basic action then amounts to changing the mental state in accordance with the transition function $\mathcal{T}$. The empty substitution $\varnothing$ is associated with a computation step due to a basic action (subscripted to $\longrightarrow$), since a basic action does not compute any bindings for variables. The use of substitutions associated with $\longrightarrow$ is explained below.

Definition 5. *(transition rules for basic actions)*

$$\frac{\mathcal{T}(\mathsf{a}(t),\sigma)=\sigma'}{\langle \mathsf{a}(t),\sigma\rangle \longrightarrow_{\varnothing} \langle E,\sigma'\rangle}$$

The semantics of tests is derived from the usual consequence relation for first order logic $\models$. Tests provide the agent with the ability to introspect its beliefs and to retrieve information from its current beliefs. The information retrieved are computed values or bindings for the free variables in the condition of a test entailed by the current belief base of the agent. The mechanism is that of logic programming. Somewhat more formal, a test $\phi?$ is a check whether or not there is an instantiation of ϕ that is entailed by the belief base of the agent. The instantiation of is computed by assigning values to the free variables in the test and results in a substitution. The bindings retrieved from the belief base thus are recorded in a substitution θ and this substitution is associated with the computation step relation $\longrightarrow$. This is because the computed bindings must be passed on to the remaining goal after the test has been executed (compare the rule for sequential composition). For example, the test $\mathit{meet}(10am, \mathit{Place})?$ where *Place* is a variable can compute the binding $\mathit{Place} = \mathit{utrecht}$ if the belief base implies $\mathit{meet}(10am, \mathit{utrecht})$.

Definition 6. *(transition rules for tests)*

$$\frac{\sigma \models \phi\theta,\, \mathit{dom}(\theta)=\mathsf{Free}(\phi)}{\langle \phi?,\sigma\rangle \longrightarrow_{\theta} \langle E,\sigma\rangle}$$

A sequential composition $\pi_1;\ \pi_2$ is executed by first executing π_1 and passing any computed bindings to π_2, which explains why the substitution θ is applied to π_2 in the rule for sequential composition below. Nondeterministic choice goals $\pi_1+\pi_2$ are executed by executing one of the subgoals π_1 or π_2 and dropping the other. Below, we only give the rule for selecting the *left* subgoal.

Definition 7. *(sequential composition and nondeterministic choice)*

$$\frac{\langle \pi_1,\sigma\rangle \longrightarrow_{\theta} \langle \pi_1',\sigma'\rangle}{\langle \pi_1;\ \pi_2,\sigma\rangle \longrightarrow_{\theta} \langle \pi_1';\ \pi_2\theta,\sigma'\rangle} \quad \frac{\langle \pi_1,\sigma\rangle \longrightarrow_{\theta} \langle \pi_1',\sigma'\rangle}{\langle \pi_1+\pi_2,\sigma\rangle \longrightarrow_{\theta} \langle \pi_1',\sigma'\rangle}$$

3 A Programming Logic for 3APL

The programming logic $\mathcal{L}$ for 3APL is an extension of the knowledge representation language used by the agents to represent their beliefs. The language $\mathcal{L}$ extends $\mathcal{L}_0$ with two modal operators. The first operator corresponds to the actions an agent can perform whereas the second is a modal operator for reasoning about the beliefs of an agent.

Definition 8. *(language $\mathcal{L}$)*
The language $\mathcal{L}$ is inductively defined by:

1. $\mathcal{L}_0 \subseteq \mathcal{L}$,
2. if $\varphi_1, \varphi_2 \in \mathcal{L}$, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}$,
3. if $x \in \mathsf{Var}$, $\varphi \in \mathcal{L}$, then $\forall\, x(\varphi) \in \mathcal{L}$,
4. if $\pi \in \mathsf{Goal}$, $\varphi \in \mathcal{L}$, then $[\pi]\varphi \in \mathcal{L}$,
5. if $\varphi \in \mathcal{L}$, then $\mathsf{B}\varphi \in \mathcal{L}$.

The notion of a sentence of $\mathcal{L}$ is somewhat more involved than the usual definition for first order language because variables in the action modality $[\pi]\varphi$ bind variables in φ. This binding corresponds to the implicit binding mechanism of the programming language based on the parameter mechanism for 3APL. Free variables in a goal π need to be instantiated during a computation and the computed bindings need to be passed on. These same values also need to be used to evaluate the conditions φ evaluated in $[\pi]\varphi$. A formula $\forall(\varphi)$ denotes the *universal closure* of formula φ, in which all free variables of φ are universally quantified.

Definition 9. (sentence)
A sentence *from $\mathcal{L}$ is defined by:*

- *if* $\mathsf{Free}(\varphi) = \{x\}$*, then* $\forall\, x(\varphi)$ *is a sentence,*
- *if* $\mathsf{Free}(\varphi) \subseteq \mathsf{Free}(\pi)$*, then* $[\pi]\varphi$ *is a sentence,*
- *if* φ_1, φ_2 *are sentences, then* $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \forall\, x(\varphi_1), [\pi]\varphi_1$ *and* $\mathsf{B}\varphi_1$ *are sentences.*

The semantics for the programming logic $\mathcal{L}$ is provided by a so-called *denotational semantics*. Below, it is proven that the *operational semantics* introduced for the programming language 3APL in the previous section is equivalent with the denotational semantics. The theorem shows how to relate an agent logic to an agent programming language, which remains unclear in, for example, [12] and [13]. Because the logical semantics is formally linked to the semantics of the programming language, properties proven in the agent logic really are properties of 3APL agent programs. The denotational semantics moreover has the important feature that it is *compositional.*

The two basic observations that are used to construct a denotational semantics are that, first, it is possible to represent the belief base of an agent by a *set of first order structures* and, second, that the usual *set of objects* in such structures can be identified with the set of constants of the knowledge representation language $\mathcal{L}_0$ since agents can only have knowledge of things which can be referred to by these constants. In effect, a *domain closure assumption* is made that every possible thing an agent can know of has a name.

Instead of the usual first order structures, we introduce the notion of a *world state* which plays a similar role in our semantics. A world state is used to specify the semantics of the belief operator B and could be used to extend our current framework to model sensory actions in the agent's environment. A world state w is a function which maps *closed* terms of the form $f(c_1, \ldots, c_n)$ to constants (the fixed domain in the semantics for $\mathcal{L}$) and maps *closed* atoms of the form $p(c_1, \ldots, c_n)$ to 1 (true) or 0 (false) where c_i are constants. This is all the information we need to interpret arbitrary *closed terms* and *sentences* from the knowledge representation language $\mathcal{L}_0$. Closed terms are interpreted by a world state as follows: $w(f(t_1, \ldots, t_n)) = w(f(c_1, \ldots, c_n))$ where $c_i = w(t_i)$ for complex terms, and $w(c) = c$ for simple terms, that is, constants. The constants in the language thus are mapped in the semantics onto themselves, and their interpretation is the same in all world states.

A set of world states is called an *epistemic state*. This terminology is justified by our first observation that a belief base of an agent can be identified with a set of world states, namely those world states which make every sentence in the belief base true. An epistemic state thus consists of the world states that are compatible with the beliefs of an agent, which is similar to the semantics for modal logics of knowledge. Epistemic states are denoted by e and the set of all epistemic states by $\mathcal{E}$.

Now we are able to define the meaning of goals in terms of these notions. Recall that basic actions are interpreted as update functions on the belief base of an agent. The semantics of basic actions therefore is represented as a function mapping one epistemic state to another since, semantically, a belief base is represented by an epistemic state. In the denotational semantics, the function $\mathcal{B}$ of type : $\mathsf{Bact} \times \mathcal{E} \rightharpoonup \mathcal{E}$ defines the semantics of basic actions and is the counterpart in the logical semantics of the function $\mathcal{T}$ of the operational semantics.

To make sure, however, that both types of semantics are equivalent, we need to impose a constraint on the transition function $\mathcal{T}$ of the operational semantics. The function $\mathcal{T}$ is directly defined in terms of the *syntactical representation* of the agent's belief base. Syntax, however, distinguishes too much and allows the possibility that two logically equivalent belief bases are updated in quite different ways by the same action. This possibility is absent in a more semantical setup as is the case with the function $\mathcal{B}$. Because a belief base in the denotational semantics is identified with a *set of world states* it is not possible to distinguish between logically equivalent belief bases as $p \wedge q$ and $q \wedge p$, for example. For this reason, we impose the following coherence constraint on the transition function $\mathcal{T}$.

Constraint. For two logically equivalent belief bases σ and σ', we require that $\mathcal{T}(\mathsf{a}(t), \sigma) = \mathcal{T}(\mathsf{a}(t), \sigma')$. That is, the effect of a basic action is not allowed to rely on the syntactical representation of the belief base of an agent.

Goals essentially perform two functions. They are update operators on the belief base of an agent and they can be used to introspect the belief base. In the former case, a new updated belief base results whereas in the latter case a set of bindings for variables may be returned. Since, semantically, we can represent a belief base by an epistemic state it is natural to interpret goals as a mapping from such epistemic states and a substitution representing the computed bindings for variables so far to (sets of) pairs consisting of two things: (i) the updated epistemic states and (ii) the newly computed bindings combined with the old ones, that is, a pair consisting of an epistemic state and a substitution. Note that goals do not change the 'current' world state, and that the denotational semantics for basic actions is well defined because of the constraint on $\mathcal{T}$ introduced above.

Definition 10. *(denotational semantics for goals)*
The interpretation function $[\![\cdot]\!]$ for goals of type : $\mathsf{Goal} \times \mathcal{E} \times \mathsf{Subs} \to \wp(\mathcal{E} \times \mathsf{Subs})$ is inductively defined by:

1. $[\![\mathsf{a}(t)]\!](e, \theta) = \{\langle \mathcal{B}(\mathsf{a}(t)\theta, e), \theta \rangle\}$,
2. $[\![\phi?]\!](e, \theta) = \{\langle e, \theta\theta' \rangle \mid \forall\, w \in e(w \models \phi\theta\theta'), dom(\theta') = \mathsf{Free}(\phi\theta)\}$,
3. $[\![\pi_1;\ \pi_2]\!](e, \theta) = \bigcup_{\langle e', \theta' \rangle \in [\![\pi_1]\!](e,\theta)} [\![\pi_2]\!](e', \theta')$, and
4. $[\![\pi_1 + \pi_2]\!](e, \theta) = [\![\pi_1]\!](e, \theta) \cup [\![\pi_2]\!](e, \theta)$.

A formal semantics for the logic $\mathcal{L}$ can be formulated using this denotational semantics for goals. Although the definition below may seem circular, since $\models$ is also used in the definition of $[\![\cdot]\!]$, this circularity can be circumvented since tests are conditions formulated in the language $\mathcal{L}_0$ and not in the full language $\mathcal{L}$ with operators $[\pi]$ for reasoning about goals. The semantics is only defined for *sentences* from $\mathcal{L}$ which allows us to do without the notion of valuation functions from traditional first order logic. Sentences from $\mathcal{L}$ are evaluated relative to a world state and an epistemic state.

A formula from the knowledge representation language $\mathcal{L}_0$ is interpreted in the current world state, as usual. The semantics of quantifiers is defined in terms of the fixed domain defined as the set of constants from $\mathcal{L}_0$. The semantics of the goal modality $[\pi]$ is provided by the denotational semantics for goals $[\![\cdot]\!]$. The belief modality is interpreted with respect to the current epistemic state; $\mathsf{B}\varphi$ is true given an epistemic state e iff all of the world states $w \in e$ entail φ (that is, all world states in e are compatible with φ).

$$
\begin{aligned}
&w, e \models p(t_1, \ldots, t_n) \text{ iff } w(p(c_1, \ldots, c_n)) = 1, \text{ where } c_i = w(t_i),\\
&w, e \models \varphi \wedge \psi \text{ iff } w, e \models \varphi \text{ and } w, e \models \psi,\\
&w, e \models \neg\varphi \text{ iff } w, e \not\models \varphi,\\
&w, e \models \forall\, x(\varphi) \text{ iff } \forall\, c \in \mathsf{Cons}(w, e \models \varphi\{x = c\}),\\
&w, e \models [\pi]\varphi \text{ iff } \forall \langle e', \theta' \rangle \in [\![\pi]\!](e, \varnothing)(w, e' \models \varphi\theta'),\\
&w, e \models \mathsf{B}\varphi \text{ iff } \forall\, w' \in e(w', e \models \varphi).
\end{aligned}
$$

3.1 Some Validities

In this section, we list a few validities. The first validity in the lemma below expresses that tests are used to *introspect the beliefs* of the agent. The other two validities are the usual axioms for sequential composition and nondeterministic choice from dynamic logic [1]. The axiom for sequential composition expresses that if after executing $\pi_1;\ \pi_2$ necessarily φ holds, then necessarily after executing π_1 we know that in case π_2 is executed consecutively φ will hold, and vice versa. The axiom for nondeterminism states that only if after execution of both π_1 *and* π_2 φ holds, we know that after executing $\pi_1 + \pi_2$ φ holds, and vice versa. Finally, the fourth validity expresses that the execution of a goal does not modify the current world state.

The following lemma is useful in proving these validities.

Lemma 1.

- $[\![\pi\theta]\!](e,\theta') = [\![\pi]\!](e,\theta\theta')$,
- $\langle e',\theta'\rangle \in [\![\pi]\!](e,\theta) \Rightarrow \theta \subseteq \theta'$.

Lemma 2.

- $\models [\phi?]\psi \leftrightarrow \forall(\mathsf{B}\phi \to \psi)$,
- $\models [\pi_1;\ \pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$,
- $\models [\pi_1 + \pi_2]\varphi \leftrightarrow ([\pi_1]\varphi \wedge [\pi_2]\varphi)$,
- $\models [\pi]\phi \leftrightarrow \phi$, for $\phi \in \mathcal{L}_0$,
- $\models [\pi]\forall\varphi \leftrightarrow \forall[\pi]\varphi$.

Proof: We prove the two first validities; the other three are left to the reader.

- $w, e \models \forall([\phi?]\psi)$ iff
  $\forall\langle e',\theta\rangle \in [\![\phi?]\!](e,\varnothing)(w, e' \models \psi\theta)$ iff
  $\forall\,\theta((\forall\, w \in e(w \models \phi\theta) \wedge \mathit{dom}(\theta) = \mathsf{Free}(\phi)) \Rightarrow w, e \models \psi\theta)$ iff
  $\forall\,\theta((w, e \models \mathsf{B}\phi\theta \wedge \mathit{dom}(\theta) = \mathsf{Free}(\phi)) \Rightarrow w, e \models \psi\theta)$ iff
  $w, e \models \forall(\mathsf{B}\phi \to \psi)$,
- $w, e \models [\pi_1;\ \pi_2]\varphi$ iff
  $\forall\langle e',\theta'\rangle \in [\![\pi_1;\ \pi_2]\!](e,\varnothing)(w, e' \models \varphi\theta')$ iff
  $\forall\langle e',\theta'\rangle \in \bigcup_{\langle e'',\theta''\rangle \in [\![\pi_1]\!](e,\varnothing)} [\![\pi_2]\!](e'',\theta'')(w, e' \models \varphi\theta')$ iff (lemma 1)
  $\forall\langle e',\theta'\rangle \in \bigcup_{\langle e'',\theta''\rangle \in [\![\pi_1]\!](e,\varnothing)} [\![\pi_2\theta'']\!](e'',\varnothing)(w, e' \models \varphi\theta''\theta')$ iff
  $\forall\langle e'',\theta''\rangle \in [\![\pi_1]\!](e,\varnothing)(w, e'' \models ([\pi_2]\varphi)\theta'')$ iff
  $w, e \models [\pi_1][\pi_2]\varphi$,

□

The first two validities below express that the B operator is normal and that an agent has positive introspection. The third validity expresses that an agent is aware of any changes in its beliefs due to executing goals. Finally, the fourth validity states that the order of the universal quantifier and the belief modality can be swapped, which is a consequence of the fact that the domain is the same in all world states.

Lemma 3.

- $\models \mathsf{B}(\varphi \to \psi) \to (\mathsf{B}\varphi \to \mathsf{B}\psi)$,
- $\models \mathsf{BB}\varphi \leftrightarrow \mathsf{B}\varphi$,
- $\models \mathsf{B}[\pi]\mathsf{B}\varphi \leftrightarrow [\pi]\mathsf{B}\varphi$,
- $\models (\forall\,\mathsf{B}\varphi) \leftrightarrow \mathsf{B}\,\forall\,\varphi$.

Proof: We only present the proof for the last validity.

$w, e \models \forall\,\mathsf{B}\varphi$ iff
$\forall\,\theta(\mathit{dom}(\theta) = \mathsf{Free}(\mathsf{B}\varphi) \Rightarrow w, e \models \mathsf{B}\varphi\theta)$ iff
$\forall\,\theta(\mathit{dom}(\theta) = \mathsf{Free}(\varphi) \Rightarrow w, e \models \mathsf{B}\varphi\theta)$ iff
$\forall\,\theta(\mathit{dom}(\theta) = \mathsf{Free}(\varphi) \Rightarrow \forall\, w' \in e(w', e \models \varphi\theta))$ iff
$\forall\, w' \in e(\forall\,\theta(\mathit{dom}(\theta) = \mathsf{Free}(\varphi) \Rightarrow w', e \models \varphi\theta))$ iff
$\forall\, w' \in e(w', e \models \forall\,\varphi)$ iff
$w, e \models \mathsf{B}\,\forall\,\varphi$

□

3.2 Equivalence of Operational and Denotational Semantics

In this subsection, we prove that the operational and denotational semantics are equivalent. The theorem below formally states this fact, and the corollary following the theorem states that the logic is suitable for proving *partial correctness properties* of agents. A partial correctness property expresses a condition on the final state in case the program finishes execution, that is, in case the agent succesfully terminates.

Theorem 1. Let e_σ ($e_{\sigma'}$) be the set of world states that satisfy σ (resp. σ').

$$\forall \sigma' (\langle \pi, \sigma \rangle \longrightarrow^*_\theta \langle E, \sigma' \rangle \text{ iff } \langle e_{\sigma'}, \theta \rangle \in [\![\pi]\!](e_\sigma, \varnothing)$$

Proof: By induction on the structure of π.

– $\pi = \mathsf{a}$: By definition.
– $\pi = \phi?$:
$\langle \phi?, \sigma \rangle \longrightarrow_\theta \langle E, \sigma \rangle$ iff
$\sigma \models \phi\theta$ and $dom(\theta) = \mathsf{Free}(\phi)$ iff
$\langle e_\sigma, \theta \rangle \in [\![\phi?]\!](e_\sigma)$.
– $\pi = \pi_1;\ \pi_2$:
$\langle \pi_1;\ \pi_2, \sigma \rangle \longrightarrow^*_\theta \langle E, \sigma' \rangle$ iff
there are σ'' and θ_1, θ_2 such that $\langle \pi_1, \sigma' \rangle \longrightarrow^*_{\theta_1} \langle E, \sigma'' \rangle$ and $\langle \pi_2\theta_1, \sigma'' \rangle \longrightarrow^*_{\theta_2} \langle E, \sigma' \rangle$ and $\theta = \theta_1\theta_2$ iff

□

The corollary shows that in case an agent believes φ after succesfully terminating execution of goal π, then this fact can be expressed in the logic $\mathcal{L}$. It also shows that in case a property $[\pi]\mathsf{B}\phi$ holds in the logic, the property really expresses a property of the program; that is, upon termination of goal π, it must be the case that the agent believes ϕ.

Corollary 1. Let e_σ be the set of world states that satisfy σ and $\phi \in \mathcal{L}_0$ a sentence from the knowledge language $\mathcal{L}_0$. Then:
$\forall \sigma' (\langle \pi, \sigma \rangle \longrightarrow^* \langle E, \sigma' \rangle \Rightarrow \sigma' \models \phi)$ iff $w, e_\sigma \models [\pi]\mathsf{B}\phi$.

Proof: Immediate from theorem 1. □

4 Conclusion

The programming logic for reasoning about a (subset of) 3APL agents presented here is a variant of a modal logic with operators for reasoning about the beliefs and the actions of an agent. As we showed, the programming logic is suitable for proving socalled *partial correctness properties* of agents. The logic is a variant of dynamic logic and can be used to prove properties of terminating 3APL agents.

In the future, we would also like to deal with nonterminating behaviours of agents. A variant of a temporal logic might be more suitable to this end.

Because 3APL *goals* are plan-like structures composed of the basic capabilities of an agent, the action modality in the logic suffices to reason about the goals and plans of the agent. In other words, there is no need for another modality to represent the goals of an agent in contrast with agent logics based on, for example, the BDI approach. From this lack of a goal modality in our programming logic we can conclude that *declarative goals* are virtually absent in the programming language 3APL. Because 3APL is a member of a family of agent languages, this same conclusion holds for other languages like AGENT0, AgentSpeak and ConGolog as well. Although we think these agent programming languages offer a viable and clear approach to building agents, we also believe that it is interesting to take declarative goals more seriously and incorporate these also into agent programming. This consideration has given rise to a new approach to agent programming called GOAL. In a complementary paper about the programming language GOAL [6], we show how to incorporate declarative goals into agent programming and provide a *temporal logic* for reasoning about GOAL agents.

A framework for proving properties of agents has been presented by providing a programming logic for 3APL agents. The logic as well as the programming language are abstract in the sense that agent capabilities are a plugin feature and need to be specified by the user. In the future, we would like to investigate in more detail the specification of these basic capabilities in the programming logic. Also, it would be interesting to investigate the effect of using other knowledge representation formalisms upon the programming logic.

Acknowledgements. We would like to acknowledge Hector Levesque for many helpful discussions about the semantics of the knowledge representation language.

References

1. David Harel. *First-order dynamic logic (LNCS 68)*. Springer-Verlag, 1979.
2. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Meyer. An Operational Semantics for the Single Agent Core of AGENT-0. Technical Report UU-CS-1999-30, Department of Computer Science, University Utrecht, 1999.
3. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL. In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence (LNAI 1502)*, pages 155–166. Springer-Verlag, 1998.
4. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
5. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Control Structures of Rule-Based Agent Languages. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI 1555)*, pages 381–396. Springer-Verlag, 1999.

6. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming with Declarative Goals. *submitted*, 2000.
7. Koen V. Hindriks, Yves Lespérance, and Hector J. Levesque. An Embedding of ConGolog in 3APL. Technical Report UU-CS-2000-13, Department of Computer Science, University Utrecht, 2000.
8. Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Meyer. Semantics of Communicating Agents Based on Deduction and Abduction. Technical Report UU-CS-1999-09, Department of Computer Science, University Utrecht, 1999.
9. Yves Lespérance, Hector J. Levesque, Fanghzen Lin, Daniel Marcu, Ray Reiter, and Richard B. Scherl. Foundations of a Logical Approach to Agent Programming. In M.J. Wooldridge, J.P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI 1037)*, pages 331–346. Springer-Verlag, 1996.
10. G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Computer Science Department, 1981.
11. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.
12. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
13. Wayne Wobcke. On the Correctness of PRS Agent Programs. In N.R. Jennings and Y. Lespérance, editor, *Intelligent Agents VI (LNAI 1757)*. Springer-Verlag, 2000.

The Nepi^2 Programming System: A π-Calculus-Based Approach to Agent-Based Programming

Yoshinobu Kawabe, Ken Mano, and Kiyoshi Kogure

NTT Communication Science Laboratories
2-4, Hikaridai, Seika-cho,
Soraku-gun, Kyoto 619-0237, Japan
{kawabe,mano,kogure}@cslab.kecl.ntt.co.jp

Abstract. We introduce a programming system Nepi^2, which is based on a process algebraic framework called the π-calculus. The Nepi^2 system supports programmers who wish to construct communicating software or agents. In this paper, we demonstrate programming in Nepi^2. First, we write a metacircular interpreter, which enables the construction of a mobile agent framework. We then construct an entity for mobile agent systems, which is called a place agent. Finally, we give an example concerning an electronic marketplace.

1 Introduction

Programming or debugging agents in a global network setting is getting more difficult due to the increasing complexity of agent communications. Programmers need to be able to foresee if an agent may cause problems before an execution in order to establish the reliability of agent-based systems. However, this foresight is very difficult because an agent's behaviors depend on the properties of other agents and so on. For example, in a mobile agent framework [1][2], a mobile agent may cause idiosyncratic problems related to resource allocation, security control, and so on; however, an agent designer cannot take into account all possible behaviors of mobile agents.

We want to develop formal techniques to check whether an agent will cause a problem or not before the performance of its operations. Checking agents' properties requires a mechanical or formal framework for verification. This framework must deal with concurrent behaviors, such as communication or a simultaneous execution of these agents. This requirement is satisfied by process algebras, which is a mathematical model for describing or analyzing distributed communication systems. Among various process algebras, the π-calculus [4][5] of Milner, Parrow, and Walker, is adequate as a formal technique for agent systems, because it is equipped with facilities for dynamic communication channel creation and inter-agent channel passing. However, there are few executable systems based on the π-calculus. Nomadic Pict [3] is a system for mobile agents based on the π-calculus, but it does not support choice, which concerns mutual exclusion.

To demonstrate how the π-calculus is used in actual programming scenes, we are developing a network programming system called Nepi^2 [7][8]. In this paper, we introduce the Nepi^2 system that enables us to directly execute concurrent process expressions in the π-calculus. The Nepi^2 system is a tool for supporting formal approaches in developing communicating software or agents. If we construct a mobile agent system or an electronic marketplace system with Nepi^2, we can execute such systems and analyze them in the π-calculus. We show several programs written in Nepi^2 and explain the expressive power of the language.

We introduce the Nepi^2 system in the next section. In sections 3 and 4, we demonstrate some programming in Nepi^2. In section 3, we describe a metacircular interpreter, and then construct a simple place agent with this interpreter. In section 4, we discuss a description of an electronic marketplace on Nepi^2.

2 The Nepi^2 Programming System

We introduce our π-calculus-based programming system Nepi^2. The Nepi^2 system is implemented in Lisp. The main features of Nepi^2 programs are the means to read and write information over channels, the creation of channels, and parallel and alternative composition of processes.

2.1 The Nepi^2 Language

Nepi^2 processes are constructed from value expressions and process expressions. Value expressions, which are data to send, are constructed from S-expressions in Lisp by adding special constants called channels. Process expressions and process definitions are constructed as in Fig. 1. From this figure, we can see that Nepi^2 processes are one of the following forms:

1. An inaction `end` that cannot perform any actions;
2. An output `(!` u `(`v`)` P`)` that writes a value expression v to a channel u. This operation is blocked until another receiver process is ready to receive the value expression;
3. An input `(?` u `(`x`)` P`)` that reads a value expression from a channel u, and then the expression substitutes all x in P. This operation is blocked until a writer process is ready to transmit;
4. A restriction `(new` ξ P`)` where the scope of channel ξ is limited to P. However, the channel may be 'extruded' from the scope by sending the channel to processes outside the scope;
5. A parallel composition `(par` P_1 P_2`)` where processes P_1 and P_2 run simultaneously. These processes can communicate with each other along some common channel. For example, we consider the following process expression:

```
(new a  (par (! a (1) end)
             (par (! a (2) end))
                  (? a (x) end))).
```

Process Expressions

P ::= `end`	Inaction
| `(new` ξ P`)`	Restriction
| `(par` P_1 P_2`)`	Parallel Composition
| `(+` $P_1 \cdots P_n$`)`	Choice
| `(!` u `(`v`)` P`)`	Output
| `(?` u `(`x`)` P`)`	Input
| `(if` v P_1 P_2`)`	Conditional
| `(`a $v_1 \cdots v_n$`)`	Process Call
| `(!std (`v`)` P`)`	Standard Output
| `(?std (`x`)` P`)`	Standard Input

Process Definitions

`(defproc` a `(`$x_1 \cdots x_n$`)` P`)` (where $i \neq j \Rightarrow x_i \neq x_j$)

Note 1: In the above definitions, ξ is a channel, $u, v, v_1, \ldots, v_n$ are value expressions, $x, x_1, \ldots, x_n$ are variable symbols, and a is a process name.

Note 2: We define non-guarded process expressions with the following grammar:

$$X ::= \texttt{(new}\ \xi\ X\texttt{)} \mid \texttt{(if}\ u\ X_1\ X_2\texttt{)} \mid \texttt{(}b\ u_1\ \cdots\ u_n\texttt{)} \mid \texttt{(par}\ Q_1\ Q_2\texttt{)},$$

where Q_1 and Q_2 are arbitrary process expressions and a process definition `(defproc` b `(`$x_1 \cdots x_n$`)` X`)` is pre-defined for b. If a process expression is not a non-guarded process expression, it is called a guarded process expression. For efficiency in implementation, we restrict $P_1, \ldots, P_n$ in a choice expression `(+` $P_1 \cdots P_n$`)` to guarded process expressions.

Fig. 1. Syntax for Nepi^2 Processes

In this case, three processes (that is, two output processes and one input process) run simultaneously. The receiver process receives either 1 or 2 from channel a; but this depends on which output process succeeds. Only one of the output processes can send a message while the other one deadlocks;

6. A choice `(+` $P_1 \cdots P_n$`)` where only one of processes $P_1, \ldots, P_n$ is executed and others are discarded. For example, we consider a process expression $X \equiv$ `(+ (!` a `(1) end) (?` a `(`x`) end))` that can send or receive a message along the channel a. Suppose that this process is embedded in a context `(new` a `(par (!` a `(2) end) _))`. In this case, the input part of X succeeds and a message 2 is sent along channel a, while the output part of X is discarded. If the X appears in another context `(new` a `(par (?` a `(`y`) end) _))`, then only the output part of X can operate the thing. As a result, a datum 1 is sent along channel a;
7. A conditional `(if` u P_1 P_2`)` that is as usual;
8. A process call `(`a $u_1 \cdots u_n$`)` that behaves as P, where the process a is specified with `(defproc` a `(`$x_1 \cdots x_n$`)` P`)`. In this operation, value expressions $u_1, \ldots, u_n$ replace variables $x_1, \ldots, x_n$ appearing in P;

9. An interaction with environments, such as a standard output or a standard input. These expressions expand the original π-calculus. A standard output `(!std` (v) P`)` writes a value v on the screen, and then behaves as P. A standard input `(?std` (x) P`)` reads a value v from a console and behaves as P, where v replaces x in P.

A Nepi^2 code is a sequence of communication actions such as `!` or `?`. A Nepi^2 process sending a message waits for a message reception from another process. Thus, the existence of a receiver process is responsible for the execution of the sender process. If a receiver process exists and accepts the message, the sender process can operate the following actions; otherwise, the action of the sender process is blocked. A sender process without the corresponding receiver cannot operate anything until a new receiver process emerges. Thus, we can say that the existence of the receiver processes limits the activity of the sender processes; this allows programmers to control the system behavior.

2.2 How Does the Nepi^2 System Work?

The Nepi^2 system has an operator for choices, while other systems such as Nomadic Pict do not. To implement synchronous communication among agents with choices, Nepi^2 has two run-time systems called the communication manager and the channel counter.

The communication manager keeps a tuple space, which is similar to a tuple space of Linda [11] or the JavaSpaces of Jini [12], to store the communication requests. Communication requests from Nepi^2 agents are first submitted to the communication manager. Then, in response to each request from a Nepi^2 agent, the manager looks for other matching communication requests in the tuple space. When such matching requests are found, the manager delivers appropriate replies to the agents that submitted these requests; otherwise, the incoming request is stored in the tuple space. The communication manager of the Nepi^2 system implements a choice operator; see [10] for details on a decentralized protocol for synchronous communication with communication managers, where it is proven that this protocol is free from deadlocks or livelocks.

Another run-time system is the channel counter, which implements the '`new`' operator. The form `(new` ξ P`)` is regarded as 'fresh channel creation' as well as restriction, where the new name ξ is replaced by another 'fresh' entity throughout a distributed system. The equivalence between the concept of restriction and that of fresh name creation is formally proved in [9]. The channel counter runs in parallel with Nepi^2 agents and the communication manager, and it issues fresh names in response to requests from Nepi^2 agents. The channel counter maintains a 64-bit counter whose value represents the next fresh name. When a Nepi^2 agent requests a fresh name, the channel counter answers with the value of the counter and increments it. Used names are not recycled. The validity of this implementation requires that the fresh names will never be exhausted as long as the system continuously operates. If we assume that the channel counter generates a fresh name every 1 nanosecond, then it exhausts all names after 584 years, which seems to be long enough.

```
(defproc mi (p d)
 (if (eq p 'end) end
 (if (eq (1st p) '!)
     (! (eval (2nd p)) ((1st (3rd p))) (mi (4th p) d))
 (if (eq (1st p) '?)
     (? (eval (2nd p)) (x) (mi (subs x (1st (3rd p)) (4th p)) d))
 (if (eq (1st p) 'par)
     (par (mi (2nd p) d) (mi (3rd p) d))
 (if (eq (1st p) 'new)
     (new x (mi (subs x (2nd p) (3rd p)) d))
 (if (eq (1st p) 'if)
     (if (eval (2nd p)) (mi (3rd p) d) (mi (4th p) d))
 (if (eq (1st p) '+)
     (+ (mi (2nd p) d) (mi (3rd p) d))
 (if (eq (1st p) '!std)
     (!std ((eval (1st (2nd p)))) (mi (3rd p) d))
 (if (eq (1st p) '?std)
     (?std (x) (mi (subs x (1st (3rd p)) (4th p)) d))
 (if (eq (1st p) 'go)
     (! (eval (2nd p)) ((cons (3rd p) d)) end)
 (mi (psubs (cdr p) (2nd (assoc (1st p) d)) (3rd (assoc (1st p) d)))
     d))))))))))))
```

Note: In this code, `eq`, `1st`, `2nd`, `3nd`, `4th`, `eval`, `cons`, `assoc` and `cdr` are system-defined Lisp functions, and '' is a Lisp macro expressing `quote`. We define user-defined Lisp functions `subs` and `psubs` which take three parameters α, β and γ to substitute β in γ to α.

Fig. 2. A Metacircular Interpreter

3 A Mobile Agent Framework Written in Nepi^2

In the rest of this paper, we demonstrate Nepi^2 programming for agent-based systems. In this section, we describe a mobile-agent framework with Nepi^2. To construct mobile-agent systems, we need a metacircular interpreter of Nepi^2.

3.1 A Metacircular Interpreter

To show the expressive power of the Nepi^2 language, we provide a metacircular interpreter of the π-calculus. Fig. 2 shows the interpreter `mi`. The process `mi` is invoked with two parameters: `p` for a process code to run and `d` for a list of process definitions. The process `mi` is defined in the same way as the function `eval` in Lisp. For example, we consider the following process expression:

```
(par (mi '(! (ch0 "well-known-port") (1) (printmes))
         '((printmes () (!std ("accepted") end))))
     (? (ch0 "well-knonw-port") (x) end)).
```

In this process, `(ch0` *string*`)` is a built-in Lisp function for generating a channel from a string. The first process of the above '`par`' is a `mi` process. The lines 3 – 4 in Fig. 2 shows that the above `mi` process becomes an output process:

```
(! (ch0 "well-known-port") (1)
   (mi '(printmes)
       '((printmes () (!std ("accepted") end))))).
```

The second process of the above '`par`' process reads a message from a channel `(ch0 "well-known-port")`. Therefore, a message '1' is transmitted. After sending the message, the first process executes:

```
(mi '(printmes) '((printmes () (!std ("accepted") end)))).
```

It expands `(printmes)`, and then interprets its body `(!std ("accepted")end)`. As a result, this process prints `"accepted"` on the screen before quitting.

Below, we use a Lisp macro:

```
(defagt a (x1 ··· xn) M)
= (setq *nepi2-defagts*
        (cons (list a (x1 ··· xn) M) *nepi2-defagts*))
```

to store process definitions of the form `(`a `(`$x_1 \cdots x_n$`)` M`)` in `*nepi2-defagts*`. We also use a Lisp function `(runagt*` P`)` to execute `(mi` P `*nepi2-defagts*)`.

3.2 Facilities for Agent Migration

Constructing a mobile-agent framework requires several technical features concerning agent migration, such as serialization or interpretation of a migrating code. We use the metacircular interpreter of Nepi^2 to interpret a migrating code. By using the metacircular interpreter, we can treat a Nepi^2 expression itself as a serialized form of a migrating agent. Moreover, the metacircular interpreter realizes strong migration, since a Nepi^2 expression represents a state of an agent as well as a program code.

The metacircular interpreter of Nepi^2 implements a new primitive construct '`go`' for agent migration (see lines 19 – 20 in Fig. 2 for the definition of '`go`'). A process expression of this primitive has a form `(go` c P`)`, where c is a channel for an identifier of a place agent and P is an expression for the body of a migrating code. The process `(go` c P`)` writes an expression `(cons` P D`)` to the channel c, where D is a list of process definitions referred by P.

To execute a migrating code `(go` c P`)`, we need a place agent that is a main component of a mobile-agent system. A place agent reads the above P from the channel c to interpret P. Fig. 3 shows a Nepi^2 code for a place agent '`place`'. A variable `id` describes an identifier of a place agent. If a Lisp object expressing a tuple `(code defs)` of a process code and a list of process definitions is sent to the place agent, a variable `prg-defs` is replaced by the object. Then, `mi` executes the mobile code.

For example, let us execute the following expression:

```
(defproc place (id)
  (? id (prg-defs)
     (par (place id)
          (mi (1st prg-defs) (2nd prg-defs)))))
```

Fig. 3. A Simple Place Agent

```
(runagt*
  '(new c
        (par (! (ch0 "well-known-port") (1)
                (go (ch0 "some-host") (out-done (ch0 "report")))
             (place (ch0 "some-host")))))),
```

where we have `(defagt out-done (x) (! x ("done") end))`. A part of this process:

```
(! (ch0 "well-known-port") (1)
   (go (ch0 "some-host") (out-done (ch0 "report"))))
```

first sends '`1`' to a channel `(ch0 "well-known-port")`. And then, this process moves to a place agent `(place (ch0 "some-host"))` for the performance of `(out-done (ch0 "report"))`. Finally, a message `"done"` is transmitted through a channel `(ch0 "report")`.

In the above example, there are no operations for an interaction with environments such as `!std` or `?std`. If we omit an agent migration, the above process is equivalent to another process:

```
(new c
     (par (! (ch0 "well-known-port") (1)
             (out-done (ch0 "report")))
          (place (ch0 "some-host")))).
```

Therefore, if a place agent `(place (ch0 "some-host"))` is running and a process `(out-done (ch0 "report"))` does not have any operation for environments, then processes `(go (ch0 "some-host") (out-done (ch0 "report")))` and `(out-done (ch0 "report"))` are equivalent in the sense of ignoring agent migration. Hence, agent migration only affects interaction with environments and does not affect inter-agent communication. If a mobile process written in Nepi^2 has no operation to access environments, then we can treat the process by omitting its migration.

3.3 An Example of Migrating Agents

We show a simple example of agent migration. Suppose that there is AGT1 on host A, while AGT2 is on host B. We describe host A, host B and the top-level process as follows.

Host A
```
(defagt host-A ()
  (par (AGT1) (place (ch0 "host-A"))))
```

Host B
```
(defagt host-B ()
  (par (AGT2) (place (ch0 "host-B"))))
```

Top-level
```
(runagt* '(par (host-A) (host-B)))
```

We give process definitions of `AGT1` and `AGT2` below.

AGT1
```
(defagt AGT1 ()
  (new myid
       (go (ch0 "host-B")
           (! (ch0 "AGT2-id") (myid)
              (! myid ('propose)
                 (? myid (x)
                    (if (equal x 'accept)
                        (go (ch0 "host-A") (proc-accept))
                      (go (ch0 "host-A") (proc-reject)))))))))
```

AGT1 first moves to the place `(place (ch0 "host-B"))` after making a local channel. It then gives the channel to AGT2 through a channel `(ch0 "AGT2-id")`, which is a well-known channel of AGT2. Next, AGT1 sends `propose` through this local channel. AGT1 waits for the response of AGT2. The response is either `'accept` or `'reject`. After getting back to the place `(place (ch0 "host-A"))`, AGT1 executes only one of `(proc-accept)` or `(proc-reject)` according to the result of the response.

We show AGT2's specification `AGT2` below, instead of showing the detailed code of AGT2. This specification is a π-calculus expression describing the communication of AGT2 with AGT1.

A Specification of AGT2
```
(defagt AGT2 ()
  (? (ch0 "AGT2-id") (local-channel)
     (? local-channel (request)
        (+ (! local-channel ('accept) end)
           (! local-channel ('reject) end)))))
```

AGT2 receives a local channel from AGT1, and it then receives a proposal with the channel. After receiving the proposal, AGT2 sends `'accept` or `'reject` back to AGT1. The entire code of AGT2 omitted here should be equivalent to this specification in the sense that there is some equivalence relation such as weak bisimilarity in CCS [6].

4 Interaction among Agents in a Marketplace

We give a larger example for agent programming in Nepi^2. An experiment on an electronic marketplace known as "the Spanish fish market" was formalized with the π-calculus framework [13]. Players, the auctioneer, buyers and scenes, admission, settlement were formalized with expressions in the π-calculus. We can explore such an auction system along with its π-calculus.

4.1 A Fish Market

There are the following main components in our marketplace.

The market boss who initiates and terminates the marketplace;
An auctioneer who declares lots and prices, arbitrates on bids;
A nomadic buyer interface (NBI) which links an auctioneer and a user-defined buyer agent to forward messages. An auctioneer should not allow user-defined agents to directly access the auctioneer. By using NBI, interfaces of an auctioneer are protected from such user-defined agents;
A buyers' manager (BM) who admits buyer agents to the market. If a user-defined buyer agent enters the market, the BM first makes an NBI and issues the NBI's identifier. The BM then gives the identifier to the buyer agent. A BM holds a list of identifiers, which is used by an auctioneer for interacting NBIs;
A seller's manager (SM) who first accepts a lot from a seller and tells the auctioneer the lot. The SM then tries to listen to the result from the auctioneer to forward it to the seller.

The market price is established through a Dutch auction, that is, a downward bidding open outcry auction.

The π-calculus description in [13] is not a program for an auction system but a specification, because several meaningless executions, as well as useful ones, are possible. For example, the auctioneer may not receive any messages from any buyers and decides that there are no buyers for a lot (see the definitions of processes `AU-StartRound`, `AU-ContinueRound` and `NBI-RelayPrice` in [13]). Hence, we cannot directly execute the specification. However, by considering how to control the system behavior of the auction system, we can make an executable Nepi^2 code from the specification. We have constructed an auction system with Nepi^2 in such a way.

4.2 The Code for an Auctioneer

Fig. 4 shows the code of an auctioneer. In this code, we use a process `scatter` defined with:

```
(defproc scatter (lst mes)
  (if (equal lst 'nil)
      end
    (! (car lst) (mes)
       (scatter (cdr lst) mes)))).
```

```
 1 (defproc AucTop (wkp-newlot BM/Auc Auc/Boss)
 2  (+ (? wkp-newlot (x)
 3        (if (eq* (car x) 'newlot) ; x = ('newlot seller lot price rp)
 4            (>> (contAuc BM/Auc (2nd x) (3rd x) (4th x) (5th x))
 5                (AucTop wkp-newlot BM/Auc Auc/Boss))
 6          (AucTop wkp-newlot BM/Auc Auc/Boss)))
 7     (? Auc/Boss (x)
 8        (! BM/Auc ('quit) end))))
 9 (defproc contAuc (BM/Auc seller lot price rp)
10  (? BM/Auc (buyers)
11     (if (<= price rp)
12         (! seller ('withdrawn)
13            (scatter buyers (list 'lose-or-withdrawn lot)))
14       (askAll BM/Auc buyers seller lot price rp 'nil))))
15 (defproc askAll (BM/Auc buyers seller lot price rp cands)
16  (if (equal buyers 'nil)
17      (checkWinner BM/Auc seller lot price rp cands)
18    (! (car buyers) ((list 'ask lot price))
19       (? (car buyers) (x)
20          (if (equal x 'no)
21              (askAll BM/Auc (cdr buyers) seller lot price rp cands)
22            (askAll BM/Auc (cdr buyers) seller lot price rp  ;x='mine
23                    (cons (car buyer) cands)))))))
24 (defproc checkWinner (BM/Auc seller lot price rp cands)
25  (if (equal cands 'nil)
26      (contAuc BM/Auc seller lot (- price 1) rp)
27  (if (equal (length cands) 1)
28      (! (car cands) ((list 'you-win lot price))
29         (! seller ((list 'winner (car cands) lot price))
30            (? BM/Auc (buyers)
31               (scatter (delete (car cands) buyers :test #'equal)
32                        (list 'lose-or-withdrawn lot)))))
33    (>> (scatter cands 'cancel)
34        (contAuc BM/Auc seller lot (floor (* price 1.25)) rp)))))
```

Fig. 4. A π-Calculus Description of An Auctioneer

We also use a new primitive '`>>`' for sequential composition. A process expression (`>>` P_1 P_2) represents that P_2 begins when and only when P_1 has finished.

An auctioneer has the following internal states of `AucTop`, `contAuc`, `askAll` or `checkWinner`.

AucTop : If an auctioneer is in this state, it non-deterministically receives a message from a SM or from the market boss. In case of SM (see lines 2 – 6 in Fig. 4), the auctioneer first checks if the message is for a new lot. If the message is valid, the auctioneer becomes '`contAuc`' to start an auction for the lot. Otherwise, the message is discarded. If the message is sent from the market boss, the marketplace is terminated (lines 7 – 8). The auctioneer sends `'quit` to terminate a BM before quitting;

contAuc : An auctioneer first obtains a list of NBIs' identifiers from a BM (line 10). The auctioneer will then check the price of a lot (line 11). If the price is less than or equal to the reservation price, then the lot should be withdrawn (lines 12 – 13). In this case, the auctioneer sends `'withdrawn` and `'lose-or-withdrawn` to a SM and all NBIs, respectively. Otherwise, the auctioneer becomes '`askAll`' (line 14);

askAll : To each NBI, an auctioneer tells the name and the price of a lot (line 18), and waits for a response from the NBI (line 19). If the response is `'no`, then the NBI does not buy the lot for the price (lines 20 – 21). Otherwise, the response should be `'mine`. In this case, this NBI is a candidate for the winner of the lot. The auctioneer has a list of such candidates, and the identifier of this NBI is stored in the list (lines 22 – 23);

checkWinner : An auctioneer checks the list for candidates. If the list is empty, nobody said `'mine` (lines 25 – 26). In this case, the price is decreased and the state of the auctioneer becomes `contAuc` to continue the auction. If the list is a singleton list, then there is only one candidate to buy the lot; the NBI is the winner (lines 27 – 32). The auctioneer declares the result to each NBI and a SM. In other cases, there are two or more candidates for the lot (lines 33 – 34). The auctioneer informs each candidate that the price is cancelled. The auctioneer then becomes '`contAuc`' with increasing the collision bid price by 25%.

4.3 Specification of a NBI

We describe a π-calculus-based specification of agents in a marketplace. Such a specification only describes the interaction with an auctioneer. We treat the case of a nomadic buyer interface (NBI). Cases of other agents are treated similarly. The following is a specification of NBI that is an expression of the π-calculus:

```
(defproc NBI (NBI/Auc)
  (? NBI/Auc (x)
     (if (equal x 'lose-or-withdrawn) (NBI NBI/Auc)
     (if (equal x 'closed) end
       (+ (! NBI/Auc ('no) (NBI NBI/Auc))
          (! NBI/Auc ('mine)
             (? NBI/Auc (reply) (NBI NBI/Auc)))))))).
```

An NBI first accepts a message from an auctioneer via a channel `NBI/Auc`. If the message is `'lose-or-withdrawn`, then the NBI executes `(NBI NBI/Auc)`. If the message is `'closed`, then the NBI quits. Otherwise, the NBI sends either `'no` or `'mine` to the auctioneer. In case of `'no`, the NBI executes `(NBI NBI/Auc)`. In case of `'mine`, the NBI receives a message before executing `(NBI NBI/Auc)`.

It seems that the NBI does nothing but accept a message if the accepted message is `'lose-or-withdrawn`. However, the NBI actually interacts with its corresponding buyer agent. If the accepted message is `'lose-or-withdrawn`, then a lot is withdrawn or another buyer is a winner for the lot. In the actual code of NBI, the NBI first reports the result to its buyer agent, and it then executes `(NBI NBI/Auc)`. The interaction of the NBI with a buyer agent is not observed by an auctioneer, so it is omitted in the specification.

The NBI specification is simple to understand. If we show that there is some equivalence relation between the actual code of NBI and its specification, such as weak bisimilarity, then we can easily treat the simplified specification rather than the entire code to understand the behavior of an agent.

5 Conclusion and Future Work

This paper has introduced a programming language Nepi^2. The Nepi^2 system is a network programming system based on the π-calculus, and we can exploit theoretical results in the π-calculus to verify the characteristics of these programs.

We demonstrated programming in Nepi^{2}by first developing a metacircular interpreter. This interpreter realizes a migrating agent in Nepi^2. We subsequently developed a mobile agent system with the metacircular interpreter.

We showed another programming example of Nepi^2, which is an electronic marketplace. We have written agents in the marketplace, such as an auctioneer, a seller or a buyer and used these agents to interact in the marketplace. We wrote a specification of a component in the marketplace called NBI. If we prove that the specification is equivalent to the actual code of NBI at a certain level of abstraction, then it is easier to understand what happens in the marketplace, such as understanding the order of communications among agents.

In the future, we want to construct tools in order to support automatic or semi-automatic verification of agents' properties with formal approaches based on the π-calculus. To provide such verification tools, we will have to develop denotational semantics of Nepi^2. In this paper, we have described an operational semantics of Nepi^2.

Acknowledgements. We would like to thank Dr. Yoh'ichi Tohkura and Dr. Katsunori Shimohara for their encouragement to our work. We also thank for Dr. Ei'ichi Horita, NTT Information Sharing Platform Laboratories, for frequent advice.

References

1. W. R. Cockayne and M. Zyda, "Mobile Agents", Manning Publications, 1997.
2. D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young and B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents", in MA 97, LNCS Vol. 1219, Springer, pp. 86-97, 1997.
3. P. Sewell, P. Wojciechowski and B. Pierce. "Location-Independent Communication for Mobile Agents: a Two-Level Architecture", Technical Report 462, Computer Laboratory, University of Cambridge, 1999.

4. R. Milner, J. Parrow and D. Walker, "A Calculus of Mobile Processes, I and II", Information and Computation, Vol. 100, pp. 1-40 and pp. 41-77, 1992.
5. R. Milner. "The Polyadic π-Calculus: a Tutorial", Technical Report ECS-LFCS-91-180, LFCS, Department of Computer Science, Univ. of Edinburgh, 1991.
6. R. Milner. "Communication and Concurrency", Prentice Hall International, 1989.
7. E. Horita and K. Mano. "Nepi^2: A Network Programming Language Based on the π-calculus", in COORDINATION 96, LNCS Vol. 1061, Springer, pp. 424-427, 1996.
8. E. Horita and K. Mano, "Nepi: A Network Programming Language Based on the π-Calculus", ECL Tech. Report, Vol. 11933, NTT Communication Science Labs., 1995.
9. Y. Kawabe, K. Mano, E. Horita and K. Kogure, "Equivalence between Restriction and Name Creation in the π-Calculus" (in Japanese), The Second JSSST Workshop on Programming and Programming Languages (PPL2000), 2000.
10. E. Horita and K. Mano, "A Decentralized Protocol for Channel-Based Communication with Choice", Tech. Report of IEICE, SS97-18, pp. 17-24, 1997.
11. D. Gelernter, "Generative communication in Linda", ACM Transactions on Program Languages and Systems 7, 1, pp. 80-112, January, 1985.
12. "The Jini Specification", http://www.sun.com/jini.
13. J. Padget and R. Bradford. "A π-calculus Model of a Spanish Fish Market — Preliminary Report —", in Agent Mediated Electronic Commerce, LNAI Vol. 1571, Springer, pp. 166-188, 1999.

From Livingstone to SMV
Formal Verification for Autonomous Spacecrafts

Charles Pecheur[1] and Reid Simmons[2]

[1] RIACS / NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
pecheur@ptolemy.arc.nasa.gov
[2] Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
reids@cs.cmu.edu

Abstract. To fulfill the needs of its deep space exploration program, NASA is actively supporting research and development in autonomy software. However, the reliable and cost-effective development and validation of autonomy systems poses a tough challenge. Traditional scenario-based testing methods fall short because of the combinatorial explosion of possible situations to be analyzed, and formal verification techniques typically require a tedious, manual modelling by formal method experts. This paper presents the application of formal verification techniques in the development of autonomous controllers based on Livingstone, a model-based health-monitoring system that can detect and diagnose anomalies and suggest possible recovery actions. We present a translator that converts the models used by Livingstone into specifications that can be verified with the SMV model checker. The translation frees the Livingstone developer from the tedious conversion of his design to SMV, and isolates him from the technical details of the SMV program. We describe different aspects of the translation and briefly discuss its application to several NASA domains.

1 Introduction

As NASA's missions continue to explore Mars and beyond, the great distances from Earth will require that they be able to perform many of their tasks with an increasing amount of autonomy, including navigation, self-diagnosis, and on-board science. For example, the Autonomous Controller for the In-Situ Propellant Production facility, designed to produce spacecraft fuel on Mars, must operate with infrequent and severely limited human intervention to control complex, real-time, and mission-critical processes over many months in poorly understood environments [4].

While autonomy offers promises of improved capabilities at a reduced operational cost, there are concerns about being able to design, implement and verify such autonomous systems in a reliable and cost-effective manner. Traditional scenario-based testing methods fall short of providing the desired confidence level, because of the combinatorial explosion of possible situations to be analyzed.

Often, formal verification techniques based on model checking[1] are able to efficiently check all possible execution traces of a system in a fully automatic way. However, the system typically has to be manually converted beforehand into the syntax accepted by the model checker. This is a tedious and complex process, that requires a good knowledge of the model chacker, and is therefore usually carried externally by a formal methods expert, rather than by the system designer himself.

This paper presents the application of formal verification techniques in the development of autonomous controllers based on Livingstone, a model-based health management and control system that helps to achieve this autonomy by detecting and diagnosing anomalies and suggesting possible recovery actions. We present a translator that converts the models used by Livingstone into specifications that can be verified with the SMV model checker from Carnegie Mellon University. The translator converts both the Livingstone model and the specification to be verified from Livingstone to SMV, and then converts any diagnostic trace from SMV back to Livingstone. It thereby shields the Livingstone application designer from the technicalities of the SMV model checker.

Sections 2 and 3 respectively present the Livingstone health management system and the SMV model checker. Section 4 introduces our translator and describes its different parts. Section 5 discusses its application to several NASA projects, Section 6 develops some comments on the nature of the verification problem for autonomy model, and Section 7 draws final conclusions.

2 Livingstone

Livingstone is a model-based health monitoring system developed at NASA Ames [9]. It uses a symbolic, qualitative model of equipment to infer its state and diagnose failures. Livingstone is one of the three parts of the Remote Agent (RA), an autonomous spacecraft controller developed by NASA Ames Research Center conjointly with the Jet Propulsion Laboratory. The two other components are the Planner/Scheduler, which generates flexible sequences of tasks for achieving mission-level goals, and the Smart Executive, which commands spacecraft systems to achieve those tasks. Remote Agent was demonstrated in flight on the Deep Space One mission (DS-1) in May 1999, marking the first control of an operational spacecraft by AI software [6]. Livingstone is also used in other applications such as the control of a propellant production plant for Mars missions and the monitoring of a mobile robot.

The functioning of Livingstone is depicted in Fig. 1. The *Mode Identification* module (MI) estimates the current state of the system by tracking the commands issued to the device. It then compares the predicted state of the device against observations received from the actual sensors. If a discrepancy is noticed, Livingstone performs a diagnosis by searching for the most likely set of component

[1] As opposed to those based on theorem proving, which can provide even more general results but require an even more involved and skilled guidance.

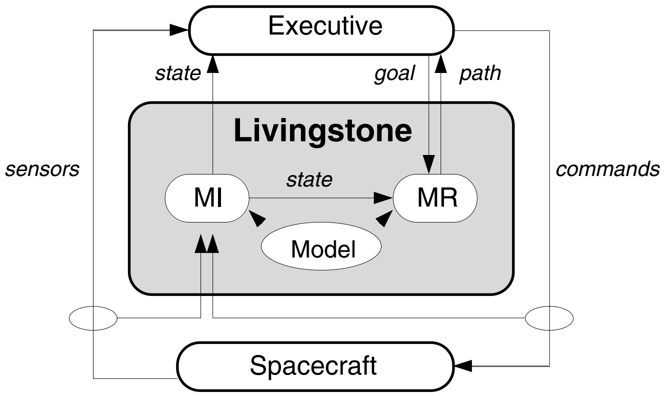


Fig. 1. Livingstone mode identification (MI) amd mode recovery (MR)

mode assignments that are consistent with the observations. Using this diagnosis, the *Mode Recovery* module (MR) can compute a path to recover to a given goal configuration.

The model used by Livingstone describes the normal and abnormal functional modes of each component in the system. Livingstone describes components using a declarative formalism called Model Programming Language (MPL), which has a Lisp-like syntax. Components are parameterized and are described using variables taking qualitative, discrete values. For each component, a set of modes is defined identifying both its nominal and failure modes. Each mode specifies constraints on the values that variables may take when the component is in that mode, and how the component can switch to other modes (by definition, spontaneous transition to any failure mode can happen from any mode). The Livingstone model thus represents a combination of concurrent finite-state transition systems. For example, Fig. 2 presents a simple MPL model for a valve, with a variable `flow` ranging over {`off`, `low`, `nominal`, `high`}. It has two nominal modes `open` and `closed` and two failure modes `stuck-open` and `stuck-closed`. The `closed` mode enforces `flow`=`off` and allows a transition `do-open` to the `open` mode, triggered when the `cmd` variable has value `open`.

3 Symbolic Model Checking and SMV

Model checking is a verification technology based on the exhaustive exploration of a system's achievable states. Given a model of a concurrent system and an expected property of that system, a model checker will run through all possible executions of that system, including all possible interleavings of concurrent threads, and report any execution that leads to a property violation.

Classical, explicit-state model checkers such as SPIN [5] do this by generating and exploring every single state. In contrast, symbolic model checking manipulates whole sets of states at once, implicitly represented as the logical conditions

```
(defvalues flow (off low nominal high))
(defvalues valve-cmd (open close no-cmd))
(defcomponent valve (?name)
  (:inputs (cmd :type valve-cmd))
  (:attributes ((flow ?name) :type flow) ...)
  (closed :type ok-mode :model (off (flow ?name))
    :transitions ((do-open :when (open cmd) :next open) ...))
  (open :type ok-mode ...)
  (stuck-closed :type fault-mode ...)
  (stuck-open :type fault-mode ...))
```

Fig. 2. A simple MPL Model of a valve

that those states satisfy. These conditions are encoded into data structures called Binary Decision Diagrams (BDDs) [1], that provide a compact representation and support very efficient manipulations. Typically, a BDD of the current set of states is combined with a BDD of the transition relation to obtain a BDD of the next set of reachable states. Symbolic model checking can address much larger systems than explicit state model checkers, but does not work well for all systems: the complexity of the BDDs can outweigh the benefits of symbolic computations, and BDDs are still exponential in the size of the system in the worst case. While symbolic model checking has traditionally been applied to hardware systems, it is increasingly being used to verify software systems, as well.

SMV, from Carnegie-Mellon University (CMU) [2], is one of the most popular symbolic model checkers. An SMV specification uses variables with finite types, grouped into a hierarchy of module declarations. Each module states its local variables, their initial value and how they change from one state to the next. Properties to be verified can be added to any module. The properties are expressed in CTL (Computation Tree Logic). CTL is a branching-time temporal logic, which means that it supports reasoning over both the breadth and the depth of the tree of possible executions. For example, the CTL formula `AG flow=high` states that **A**lways (for all executions) **G**lobally (all along each execution), the flow is high.

4 Verification of Livingstone Models

The Livingstone engine performs complex computations using large-size data structures capturing its knowledge about the model. In order to apply model checking to the autonomous controller as a whole, we would need an SMV specification of the Livingstone engine and its data structures, including the Livingstone model. Producing such a specification would be an arduous and error-prone task. Furthermore, the size of the data structures involved would severely limit the tractability of model checking.

Alternatively, the autonomy model can be considered as a high-level program that is “executed”, in a somewhat unusual way, by Livingstone. The Livingstone

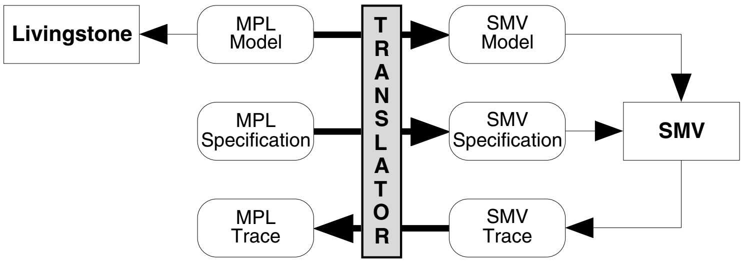


Fig. 3. Three kinds of translation between MPL and SMV

program itself is a more complex, but also more stable and better understood part, built around well-documented algorithms. Since it remains basically unchanged between applications, the verification of its correctness can be done once and for all by its designers and is not addressed here. From the point of view of its users, Livingstone is viewed as a stable, trustable part, just as C programmers trust their C compiler.

Therefore, the focus of this article is on the verification of a Livingstone model with respect to the real system that this model represents. This can be addressed by turning this model into a representation suitable for model checking. Since this model is specific to the application that it is used for, it is indeed where the correctness issues are most likely to occur.

In many previous experiences in model checking of software, this translation has been done by hand. This is usually the most complex and time-consuming part, typically taking weeks or months, whereas the running the verification is a matter of minutes or hours thanks to the processing power of today's computers. The net result is that software model checking is currently mostly performed off-track by formal methods experts, rather than by field engineers as part of the development process.

Our goal is to allow Livingstone application developers to use model checking to assist them in designing and correcting their models, as part of their usual development environment. To achieve that, we have developed a translator to automate the conversion between MPL and SMV. To completely isolate the Livingstone developer from the syntax and technical details of the SMV version of his model, we need to address three kinds of translation, as shown in Fig. 3:

- The MPL model needs to be translated into an SMV model amenable to model checking.
- The specifications to be verified against this model need to be expressible in terms of the MPL model and similarly translated.
- Finally, the diagnostic traces produced by SMV need to be converted back in terms of the MPL model.

```
MODULE valve
VAR     mode: {open,closed,stuck-open,stuck-closed};
        cmd: {open,close,no-cmd};
        flow: {off,low,nominal,high};
DEFINE faults:={stuck-open,stuck-closed};
INVAR mode=closed -> flow=off
TRANS (mode=closed & cmd=open) ->
        (next(mode)=open | next(mode) in faults)
```

Fig. 4. SMV Model of a Valve

These three aspects are covered by our translator and are detailed in the following sub-sections. The translator program has been written in Lisp[2] and is about 4000 lines long.

4.1 Translation of Models

The translation of MPL models to SMV is facilitated by the strong similarities between Livingstone models and SMV specifications. In particular, both have synchronous concurrency semantics. The main difficulty in performing the translation comes from discrepancies in variable naming rules between the flat name space of Livingstone and the hierarchical name space of SMV. Each MPL variable reference, such as `(flow valve-1)`, needs to be converted into a SMV qualified variable reference w.r.t. the module hierarchy, e.g. `ispp.inlet.valve-1.flow`. To optimize this process, the translator builds a lexicon of all variables declared in the MPL model with their SMV counterpart, and then uses it in all three parts of the translation. Figure 4 presents the SMV translation of the MPL model in Figure 2.

4.2 Translation of Specifications

The specifications to be verified with SMV are added to the MPL model using a new `defverify` declaration[3]. The `defverify` declaration also defines the top-level module to be verified. The properties to be verified are expressed in a Lisp-like style that is consistent with the rest of the MPL syntax; their translation function is an extension of the one used for MPL's logic formulae. For example, the declaration in Fig. 5 specifies a CTL property to be verified on module `ispp`. Without entering into details of CTL, the specification says that, from any non-failure state, a high flow in valve 1 can eventually be reached. Fig. 6 shows the top-level SMV module that is produced from that declaration.

In SMV, specifications use the powerful temporal logic CTL. CTL is very expressive but requires a lot of caution and expertise to be used correctly. To

[2] Lisp was a natural choice considering the Lisp-style syntax of the MPL language.

[3] This is specific to the translator and rejected by Livingstone; an added empty Lisp macro definition easily fixes this problem.

```
(defverify
  (:structure (ispp))
  (:specification
     (always (globally (implies
       (not (broken))
       (exists (eventually (high (flow valve-1)))))))
```

Fig. 5. Specification for verification in MPL

```
MODULE main
  VAR ispp : ispp;
  SPEC AG ((!broken) ->
    EF (ispp.inlet.valve-1.flow = high))
```

Fig. 6. Specification for verification in SMV

alleviate this problem, the translator supports several alternative ways of expressing model properties.

Plain CTL. CTL operators are supported in MPL's Lisp-like syntax, as illustrated in Fig. 5.

Specification Patterns. Common properties such as reachability of given component modes can be concisely expressed using pre-defined specification patterns such as `(:reachability (valve valve-1))`.

Consistency and completeness are a prime source of trouble for designers of Livingstone models. For example, for all transition statements `(name :when <cond> :next <mode>)` associated to the same mode, it is required that exactly one of the conditions `<cond>` hold at each step. If two transitions are enabled simultaneously, then two next modes are enforced at the same time, resulting in inconsistency. To catch these problems, the specification pattern `(:disjointness <comp>)` extracts the guards of all transitions of component `<comp>` in the model and generates, for each mode, a mutual exclusion property among its transitions. A peer pattern `(:completeness <comp>)` checks that at least one guard is always fulfilled.

Auxiliary Functions. The translator supports some auxiliary functions that can be used in CTL formulas to concisely capture Livingstone concepts such as occurrence of failures, activation of commands or probability of failures. Table 1 gives a representative sample. Some functions are translated solely in terms of SMV logic expressions, while others, such as `failed`, require the introduction of new variables[4].

[4] The latter are omitted by default, since the new variables can cause a big penalty on the performance of SMV.

Table 1. Some auxiliary functions for MPL model specifications

`(broken heater)`	=	Heater is in a failed state.
`(failed heater)`	=	On last transition, heater failed.
`(multicommand 2)`	=	At least two commands are activated.
`(brokenproba 3)`	=	Combined probability of currently failed components is at most of order 3.

The probability analysis opens an interesting perspective. Failure probabilities are mapped to small integer order-of-magnitude values (e.g. $p = 10^{-3}$ maps to 3), so that the value for multiple failures can be computed by integer addition, which is supported by SMV's BDD-based analysis. One should note, however, that this is an approximate method, which fits well with the qualitative nature of Livingstone models but is no substitute for a precise approach such as Markov chain analysis.

Observers. The translator allows the definition of *observer automata*, which are cross-breeds between modules (in that they can refer to other components or modules) and components (in that they can have modes). An observer, however, can have no internal variables, other than keeping track of mode. Observers are useful in some situations where the CTL specification language is inadequate for representing the specifications that one wants to verify.

4.3 Translation of Traces

When a violated specification is found, SMV reports a diagnostic trace, consisting of a sequence of states leading to the violation. This trace is essential for diagnosing the nature of the violation.

The states in the trace, however, show variables by their SMV names. To make sense to the Livingstone developer, it is translated back in terms of the variables of the original MPL model. This is achieved using the lexicon generated for the model translation in the reverse direction.

A more arduous difficulty is that the diagnostic trace merely indicates the states that led to the violation but gives no indication of what, within those states, is really responsible. Two approaches to this diagnosis problem are currently being investigated. One is based on using visualization tools to expose the trace, the other one uses a truth maintenance system to produce causal explanations [8].

5 Applications

5.1 Deep Space One

Livingstone was originally developed to provide model-based diagnosis and recovery for the Remote Agent architecture on the DS1 spacecraft. The full Livingstone model for the spacecraft runs to several thousand lines of MPL code.

Using the translator, we have automatically constructed SMV models and verified several important properties, including consistency and completeness of the mode transition relations, and reachability of each mode. We are also developing specialized declarations to enable us to verify path reachability properties, such as the ability of the system to transition from a fault mode to a known "safe" mode. Using the translator, we were able to identify several (minor) bugs in the DS1 models (this was after the models had been extensively tested by more traditional means) [7].

5.2 ISPP

The translator is being used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP), a system that will produce spacecraft propellant using the atmosphere of Mars [3]. First experiments have shown that SMV can easily process the ISPP model and verify useful properties such as reachability of normal operating conditions or recoverability from failures. The current version of the ISPP model, with 10^{50} states, can still be processed in less than a minute using SMV optimizations (re-ordering of variables). The Livingstone model of ISPP features a huge state space but little depth (all states can be reached within at most three transitions), for which the symbolic processing of SMV is very appropriate.

6 Lessons Learned

Concrete applications have shown that the nature of the verification problem for Livingstone models is quite distinct from the verification of a more conventional concurrent application. A typical concurrent system is a collection of active entities, each following a well scoped algorithm. In contrast, a typical Livingstone module describes a passive component such as a tank, valve or sensor; it states how this component reacts to external commands but hardly ever imposes any kind of order of operations in the component itself. On top of that, failures amount to unrestricted spontaneous transitions in every component that allows them.

This results in state spaces that have a very peculiar shape: a huge branching factor, due to all the command variables that can be set and all the failures that can occur at any given step, but a very low depth, due to the very little inherent sequential constraints in the model. In other words, a typical reachability tree for an MPL model is very broad but very shallow, with every state reachable from the initial one within a few transitions.

This also affects the kind of properties that are useful to verify. Looking for deadlocks makes no sense in the presence of spontaneous failure transitions, though more focused reachability properties can reveal inconsistencies in the model. More typically, though, one is interested in consistency and completeness properties, because the declarative nature of MPL makes it very easy to come up with an over- or under-constrained model.

7 Conclusions

Our MPL to SMV translator allows the Livingstone-based application developers to take their MPL model, specify desired properties in a natural extension of their familiar MPL syntax, use SMV to check them and get the results in terms of their MPL model, without reading or writing a single line of SMV code. This kind of separation is an important step towards a wider adoption of verification methods and tools by the software design community.

SMV seems to be very appropriate for certifying Livingstone models for several reasons. First of all, the Livingstone and SMV execution models have a lot in common; they are both based on conditions on finite-range variables and synchronous transitions. Second, the BDD-based symbolic model checking is very efficient for such synchronous systems and appears to fit well to the loosely constrained behaviors captured by Livingstone models. Due to this good match and to the high level of abstraction already achieved by the Livingstone models themselves, it is possible to perform an exhaustive analysis of a direct translation of those models, even for fairly complex models. In contrast, more conventional software model checking applications almost always require some abstraction and simplification stage to make the model amenable to model checking.

This work shows that verification of Livingstone models can be a useful tool for improving the development of Livingstone-based applications. It is, however, only one piece in the larger problem of building and validating autonomous applications. It cannot establish that the Livingstone mode identification will properly identify a situation (though it can establish that there is not enough information to do it). Neither does it address the interaction of Livingstone with other parts of the system, including real hardware with hard timing issues. Other complementary approaches are needed. In this line of work, we are currently prototyping an analytic testing approach based on a controlled execusion of an instrumented version of the real Livingstone program in a simulated environment.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, **C-35(8)**, 1986.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. Information and Computation, **98(2)**, June 1992, pp. 142-170.
3. D. Clancy, W. Larson, C. Pecheur, P. Engrand and C. Goodrich. Autonomous Control of an In-Situ Propellant Production Plant. Technology 2009 Conference, Miami, November 1999.
4. A. R. Gross, K. R. Sridhar, W. E. Larson, D. J. Clancy, C. Pecheur, and G. A. Briggs. Information Technology and Control Needs For In-Situ Resource Utilization. Proceedings of the 50th IAF Congress, Amsterdam, Holland, October 1999.
5. G. J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, **23(5)**, May 1997.

6. N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. Artificial Intelligence **103(1-2)**:5-48,August 1998.
7. P. P. Nayak et al. Validating the DS1 Remote Agent Experiment. In: Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS-99), ESTEC, Noordwijk, 1999.
8. R. Simmons and C. Pecheur. Automating Model Checking for Autonomous Systems. AAAI Spring Symposium on Real-Time Autonomous Systems, March 2000.
9. B. C. Williams and P. P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. Proceedings of AAAI-96, 1996.

Verification of Plan Models Using UPPAAL

Lina Khatib, Nicola Muscettola, and Klaus Havelund

NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035
QSS Group, Inc. and RECOM Technologies
{lina,mus,havelund}@ptolemy.arc.nasa.gov

Abstract. This paper describes work on the verification of HSTS, the planner and scheduler of the Remote Agent autonomous control system deployed in Deep Space 1 (DS1)[8]. The verification is done using UPPAAL, a real time model checking tool [6]. We start by motivating our work in the introduction. Then we give a brief description of HSTS and UPPAAL. After that, we give a mapping of HSTS models into UPPAAL and we present samples of plan model properties one may want to verify. Finally, we conclude with a summary.

Keywords. Model Checking, Verification, Autonomy, Planning, Scheduling

1 Introduction

AI technologies, and specifically AI planning, facilitates the elicitation and automatic manipulation of system level constraints. However, the models used by the planner still need to be verified, i.e., it is necessary to guarantee that no unintended consequences will arise. One question that comes to mind is whether the most advanced techniques used in software verification, specifically model checking, can help.

The most used model checking formalisms, however, cannot easily represent constraints that are naturally represented by HSTS, namely continuous time and other continuous parameters. Also, the goal of HSTS is to provide an expressive language to facilitate knowledge acquisition by non AI experts (e.g., system engineers). So, HSTS models cannot be easily translated into a model checking formalism. To allow model checking algorithms to operate on HSTS models we, therefore, need a mapping between a *subset* of the HSTS Domain Description Language to a model checking formalism. An earlier attempt to analyze HSTS planner models, where no continuous parameters were considered, is described in [10].

We choose UPPAAL because it can represent time (section 3), and is comparable to HSTS in terms of representation and search since they are both constraint based systems. Furthermore, UPPAAL has been successfully applied to several verification cases of real-time systems of industrial interest [4,3].

Some of the issues that we are interested in addressing in this research are:

1. Whether model checking techniques can address problems of the size of a realistic autonomy planning application;
2. Once we have a mapping from a planner model into a model checking formalism, what the core differences between the search method used in model checking and that used by a planner are; and
3. Lessons, if any, that planning can take from model checking and vice versa regarding representation, search control, and other aspects. Related work can be found in [1,2].

2 HSTS

HSTS, Heuristic Scheduling Testbed System, is a general constraint-based planner and scheduler. It is also one of four components that constitutes the Remote Agent Autonomous system which was used for the Remote Agent Experiment (RAX), in May of 1999, for autonomous control of the Deep Space 1 spacecraft [8].

HSTS consists of a planning engine that takes a plan model, in addition to a goal, as input and produces a complete plan. A plan model is a description of the domain given as a set of objects and constraints. The produced plan achieves the specified goal and satisfies the constraints in the plan model.

An HSTS plan is a complete assignment of tokens for all the state variables that satisfy all compatibilities. HSTS ensures robustness of schedules by allowing for flexible temporal representations, *ranges* of durations, and disjunction of constraints [9,5].

2.1 HSTS Model

An HSTS plan model is specified in an object-oriented language called DDL (Domain Description Language). It is based on a collection of objects that belong to different classes. Each object is a collection of state variables (also known as timelines). At any time, a state variable is in one, and only, one state that is identified by a predicate. Tokens are used to represent "spans", or intervals, of time over which a state variable is in a certain state.

A set of compatibilities between predicates is specified. The compatibilities are temporal constraints, which may involve durations between end points of tokens. For example, "token1 meets token2" indicates that the end of token1 should coincide with the start of token2; and "token1 before[3,5] token2" indicates that the distance between the end of token1 and the start of token2 is between 3 and 5. The compatibilities among tokens are structured in the form of a master, the constrained token, and servers, the constraining tokens, and stated in the form of AND/OR trees (see the Rover and Rock example for illustration). HSTS has a rich language for expressing temporal relation constraints which is beyond the scope of this paper. Details can be found in [9,5].

HSTS also allows for natural and efficient handling of concurrent processes, and the modeling language is simple in its uniform representation of actions and states. The following example will be used for illustration throughout the paper.

```
State Variables: Rover, Rock
Rover predicates: atS, gotoRock, getRock, gotoS
Rock predicates: atL, withRover

Compats:
 1. Rover.getRock   dur[3, 9]
        AND
           metby Rover.gotoRock
           meets Rock.withRover
           OR
               meets Rover.gotoS
               meets Rover.gotoRock
 2. Rover.atS   dur[0, 10]
        AND
           metby Rover.gotoS
           meets Rover.gotoRock
 3. Rover.gotoRock   dur[5, 20]
        AND
           OR
               metby Rover.atS
               metby Rover.getRock
           meets Rover.getRock
 4. Rover.gotoS
        AND
           metby Rover.getRock
           meets Rover.atS
 5. Rock.atL
        meets Rock.withRover
 6. Rock.withRover
        metby Rover.getRock
```

Fig. 1. DDL Model for the Rover and Rock Example

Example (Rover and Rock). Figure 1 shows an HSTS model, in abstract syntax, that describes the domain of a Rover that is to collect samples of Rocks. In this example, there are two objects, the Rover and the Rock, each of which consists of a single state variable. The Rover's state variable has a value domain of size 4 which includes atS, gotoRock, getRock, and gotoS (where "S" stands for Spacecraft). The Rock's state variable has a value domain of size 2 which includes atL and withRover ("L" is assumed to be the location of the Rock). Each predicate is associated with a set of compatibilities (constraints). We choose to explain the compatibilities on Rover.getRock, for the purpose of illustration. A token representing the predicate Rover.getRock should have a duration no less than 3 and no more than 9. It also have to be preceded immediately by Rover.gotoRock and followed immediately by Rock.withRover. The last compatibility indicates that Rover.getRock should be followed immediately by either Rover.gotoS or Rover.gotoRock (to pickup another rock).

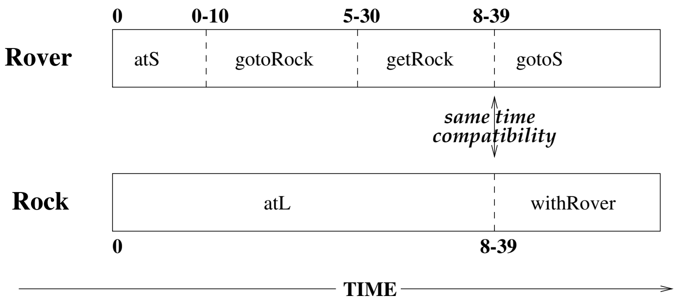


Fig. 2. An HSTS plan for the goal of Rock.withRover given the initial state (Rover.atS,Rock,atL). Dashed lines indicate token boundaries. The numbers at boundaries indicated ranges of earliest and latest execution times. A constraint link is attached between the end of Rover.getRock and the start of Rock.withRover to insure the satisfaction of their equality constraint.

Figure 2 shows a plan generated by HSTS for a given goal. In this example, the initial state is: Rover.atS and Rock.atL and the goal is to have Rock.withRover. The returned plan for Rover is the sequence atS, gotoRock, getRock, and gotoS where the allowed span for each of these tokens is as specified in the duration constraints of their compatibility (e.g., getRock token is between 3 and 9 time units). As a result, the goal of Rock.withRover may be satisfied (executed) in 8 to 39 time units. The quality of generated plans is improved by interleaving planning and scheduling, rather than performing them separately.

3 UPPAAL

UPPAAL, an acronym based on a combination of UPPsala and AALborg universities, is a tool box for modeling, simulation, and verification of real-time systems. The simulator is used for interactive analysis of system behavior during early design stages while the verifier, which is a model-checker, covers the exhaustive dynamic behavior of the system for proving safety and bounded liveness properties. The verifier, which is a symbolic model checker, is implemented using sophisticated constraint-solving techniques where efficiency and optimization are emphasized. Space reduction is accomplished by both local and global reduction. The local reduction involves reducing the amount of space a symbolic state occupies and is accomplished by the compact representation of Difference Bounded Matrix (DBM) for clock constraints. The global reduction involves reducing the number of states to save during a course of reachability analysis [11, 7].

A UPPAAL model consists of a set of timed automata, a set of clocks, global variables, and synchronizing channels. A node in an automaton may be associated with an invariant, which is a set of clock constraints, for enforcing transitions out of the node. An arc may be associated with a guard for controlling when this transition can be taken. On any transition, local clocks may get reset and global variables may get re-assigned. A trace in UPPAAL is a sequence of states, each of which containing a complete specification of a node from each automata, such that each state is the result of a valid transition from the previous state.

UPPAAL had been proven to be a useful model_checking tool for many domains including distributed multimedia and power controller applications [4,3].

4 UPPAAL for HSTS

Figure 3 shows the overall structures of UPPAAL (represented as Model Checking) and HSTS (represented as Planning). There is an apparent similarity between their components. Model checking takes a model and a property as input and produces the truth value of the property in addition to a diagnosis trace. Planning takes a model and a goal as input and produces a complete plan that satisfies the goal. On the other hand, the representation and reasoning techniques for their components are different. Table 1 summarizes the differences.

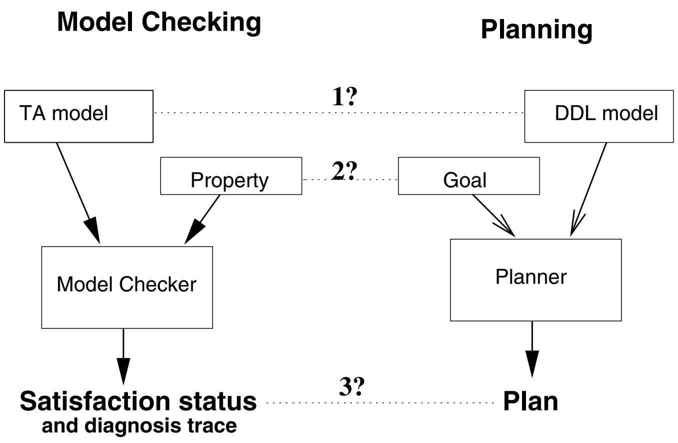


Fig. 3. Model Checking and Planning Structures. The dotted arrows represent possible component interchangeability.

Due to structural similarity of UPPAAL and HSTS, a cross fertilization among their components may be possible. Also, due to the differences in their implemented techniques, this may be fruitful. Our research at this time is to investigate the benefit of using the UPPAAL reasoning engine to verify HSTS models as well as verifying the HSTS reasoning engine. The first step is to find a mapping from HSTS models into UPPAAL. Then, a set of properties should be carefully constructed and checked.

Table 1. Representation and Reasoning of UPPAAL and HSTS. DBM stands for Difference Bounded Matrices and CDD stands for Clock Difference Diagrams.

	UPPAAL	HSTS
INPUT Domain Model: Problem:	 Timed Automata Check Property	 DDL Accomplish Goal
OUTPUT Solution:	 Property status + Diagnosis trace	 A Plan
INNER WORK Database: Search Techniques:	 DBM or CDD + explored states Forward Reachability + Shortest Path Reductions	 Temporal and Constraint networks Propagations + Heuristics directed backtracking

```
DDL2UPPAALmain()
1. Build_Init_Automata() ;
      for each State Variable, add an Automaton with a dedicated local clock
      for each predicate, add a node in the corresponding automaton
      for each node, reset the local clock on the outgoing arc
2. Add_Compatibilities();
      for each compatibility on a predicate corresponding to a node P
            for max duration constraint, add invariant on P
            for min duration constraint, add a guarded outgoing arc from P
            Process the AND/OR compatibility tree
                  if (root = "AND") process AND-subtree(root)
                  elseif (root = "OR") process OR-subtree(root)
                  else process simple-Temporal-Relation(root)
```

Fig. 4. ddl2uppaal: An algorithm for mapping HSTS models into UPPAAL

4.1 Mapping HSTS Models into UPPAAL

An algorithm for mapping HSTS plan models into UPPAAL models, which is called ddl2uppaal, is presented in Figure 4. Each state variable is represented as a UPPAAL automaton where each value of the state variable is represented as a node. Transitions of an automaton represent value ordering constraints of the corresponding state variable. Duration constraints are translated into invariants and guards of local clocks. Temporal relation constraints are implemented through communication channels. In general, constraints on the starting point of a predicate are mapped into conditional *incoming* arcs into its node. Similarly, constraints on the end point of a predicate are mapped into conditional *outgoing* arcs from its node. Instead of presenting the lower level details of the mapping algorithm, we choose to illustrate them through the example. For this purpose, we apply ddl2uppaal on the Rover and Rock specification and show the results in Figure 5. Studying Rover.getRock node, we find the duration constraint represented as the c1<=9 invariant and the c1>= 3 guard on the outgoing arc. The constraint of metby Rover.gotoRock is represented by the incoming arc. The constraint of (meets Rover.gotoS **OR** meets Rover.gotoRock) is represented by branching outgoing arc. Finally, the constraint of meets Rock.withRover is expressed via the label 'ch1?' on its outgoing arc, which indicates a need for synchronization with a transition labeled with 'ch1!'. This transition is the incoming arc to Rock.withRover.

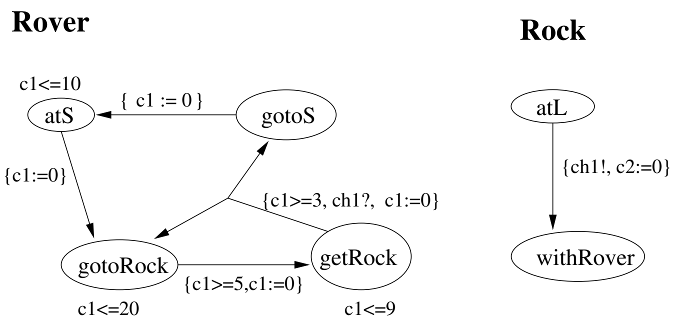


Fig. 5. UPPAAL model of the Rover and Rock example. c1 is the local clock of Rover and c2 is the local clock of Rock.

4.2 Properties for Verification

UPPAAL allows for verifying properties that are useful for ensuring correctness and detecting inconsistencies and flaws in HSTS plan models. For example, UPPAAL is capable of detecting violations of mutual exclusion properties of predicates, which is useful for detecting an incomplete specification of compatibilities in an HSTS model. Also, from checking the reachability of predicates, inconsistencies in an HSTS model may be detected.

```
Initial state: (Rover.atS, Rock.atL)
Property: E<>Rock.withRover

OUTPUT: PROPERTY IS SATISFIED
Diagnosis Trace:
        (Rover.atS, Rock.atL)
        (Rover.gotoRock, Rock.atL)
        (Rover.getRock, Rock.atL)
        (Rover.gotoS, Rock.withRover)
```

Fig. 6. Verifying Property in UPPAAL (Example).

Goals in HSTS can be mapped into properties in UPPAAL and execution traces in UPPAAL correspond to plans in HSTS. Figure 6 shows a UPPAAL property and diagnosis trace that correspond to the HSTS goal and plan of Figure 2. Note the generated symbolic states in the trace and compare them to the tokens of the plan.

Based on the above, UPPAAL is able to verify the existence of complete plans that satisfy given constraints. This can be used to verify HSTS models as well as verifying the HSTS engine.

5 Summary

Our work tackles the problem of using Model Checking for the purpose of verifying planning systems.

We presented an algorithm that maps plan models into timed automata. The algorithm works well for translating models of limited size and complexity. Since complete constraint planning models are much too complex for a complete translation into a model checking formalism, there is a need for building representative "abstract" models. We will investigate such abstraction in the near future.

After translating a plan model, properties can be checked for detecting inconsistencies and incompleteness in the model. In addition, the model checking search engine can be used as an independent problem solving mechanism for verifying the planning engine. This is possible because goals can be mapped into properties and traces correspond to plans. We have illustrated such correspondence through an example.

We are currently working on identifying a set of verification properties that guarantee a certain degree of coverage for HSTS models and the Planning engine. We are also analyzing the benefits, and limitations, of using a model checker for HSTS verification. In addition, we are extending the ddl2uppaal algorithm to handle a larger subset of DDL.

Acknowledgment. The authors would like to thank the UPPAAL team, especially Paul Pettersson, for their helpful correspondences.

References

1. A. Cimatti, M. Roveri, and P. Traverso. 1998. Strong planning in non-deterministic domains via model checking. In the Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS98), pp. 36-43. AAAI Press.
2. M. Di Manzo, E. Giunchiglia, and S. Ruffino. 1998. Planning via model checking in deterministic domains: Preliminary report. In the Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA98), pp. 221-229. Springer-Verlag.
3. K. Havelund, K. G. Larsen, and A. Skou. 1999. Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL. In the Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems.

4. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. 1997. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In the Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 14-24. San Francisco, California.
5. A. K. Jonsson, P. H. Morris, N. Muscettola, and K. Rajan. 1999. Planning in Interplanetary Space: Theory and Practice. American Association for Artificial Intelligence (AAAI-99)
6. K. G. Larsen, P. Pettersson, and W. Yi. 1997. UPPAAL in a Nutshell In Springer International Journal of Software Tools for Technology Transfer 1(1+2).
7. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. 1997. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In the Proceedings of the 18th IEEE Real-Time Systems Symposium, pages 14-24. IEEE Computer Society Press.
8. Muscettola, P. P. Nayak, B. Pell, and B. William. 1998 Remote Agent: To boldly go where no AI system has gone before. Artificial Intelligence 103(1-2):5-48
9. N. Muscettola. 1994. HSTS: Integrated planning and scheduling. In M. Zweben and M. Fox, eds., Intelligent Scheduling. Morgan Kaufman. 169-212
10. J. Penix, C. Pecheur, K. Havelund. 1998. Using Model Checking to Validate AI Planner Domain Models. In the Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard.
11. W. Yi, P. Pettersson, and M. Daniels. 1994. Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, Proceedings of the 7th International Conference on Formal Description Techniques, pages 223-238. North-Holland.

Formalization of a Spatialized Multiagent Model Using Coloured Petri Nets for the Study of an Hunting Management System

Innocent Bakam[1], Fabrice Kordon[2], Christophe Le Page[3], and François Bousquet[3]

[1] Mathematics and Computer Science Department, Faculty of Science,
Douala University, Po Box 2701 Douala, Cameroun.
`ibakam@uycdc.uninet.cm`

[2] Laboratoire d'Informatique de Paris 6/SRC,
Université Paris VI, 4 place Jussieu, 75252 Paris Cedex 05, France.
Fabrice.Kordon@lip6.fr

[3] CIRAD-TERA/ERE,
Campus International de Baillarguet,
BP 5035 34032 Montpellier, France.
`{lepage, bousquet}@cirad.fr`

Abstract. This paper presents an experience of a multiagent model formalization using coloured Petri nets, applied to the study of an hunting management system. The multiagent model of the hunting activity is presented, with simulation results. Then we describe Petri nets model assumptions and we give details about system modules. We analyze the model and we compare formal properties to multiagent simulation results. Pertinence of the approach is briefly discussed

1 Introduction

Multiagent approach provides a new way to model phenomena in which interactions between various entities are too complex to be apprehended by the traditional tools of mathematical modelling. Thus this approach is increasingly used in problems of environment management, and particularly in modelling of natural and renewable resources management [1], [2], [3]. The expression power of multiagent models allows us to represent interactions between autonomous entities, which have an individual behavior and are able to evolve in an environment [8]. But when it is necessary to analyze the system and identify global and general properties, the multiagent approach appears to have limits. The use of formal methods of specification seems to be the way indicated to analyze these virtual worlds which can not be studied only by simulation.

We present in this paper an experiment of the use of a formal approach to apprehend a multiagent system. Singh [13] underlines that althought several powerful formalisms exist, finding the right formalism is a nontrivial challenge. From a multiagent model of the hunting activity based on data collected in an

Eastern Cameroon village, we have built an equivalent Petri net model. Petri nets are choisen as a middle way between two extremes: differential equations which are not adapted to model individual behaviours and multiagent approach which has inadequacies when trying any formalization. After some simplifications, we use a model checker to exhibit formal properties of the system. Some previous related works have already attemped to use Petri nets to formalize multiagent models [4], [5] but these works usually focus on the behaviour aspect of those agents. Since in the field of natural and renewable resources modelling the space is often very important, we have built a Petri net model which integrates spatial distribution of the agents and their moves in this space. Gronewold [6] gives an example how the modelling technique of Petri nets can be adopted to the individual oriented modelling of ecological systems. But they are inadequacies in that work, according to the possibility to demonstrate general properties of the model.

We will begin with a brief presentation of the multiagent model, after which we will give the coloured Petri net we have built there from and the analysis we have carried on this model using available tools. We will discuss the pertinence of the approach and we will end by drawing some conclusion on the use of formal methods according to our experiment.

2 The Multiagent Model of Hunting Activity

During these last years, different organizations concerned with wildlife are unanimous that African fauna is increasingly being destroyed by anarchical hunting pratices. Many solutions (protected areas, taxes) have been tested, without producing the expected effects. The purpose of the use of multiagent approach to model hunting activity was to evaluate the viability of local strategies of wildlife preservation. The multiagent model we have built [9] represents hunting of the blue duiker in the forest around the village. An artificial landscape, mapping on the forest landscape has been defined as a grid of cells. Each cell has attributes corresponding to the state of the space portion it represents : road, water, trap...

A duiker agent has been created. Its attributes are the age, the sex, the duration of gestation and the partner. Using data on the life history of the blue duiker obtained from the work of Dubost [7], we have implemented growth, mortality and reproduction functions.

Then we have simulated the natural evolution of the population without any human activity. The results for population density suggest that the multiagent model converges with damping oscillations to a steady state of approximately 90 animals per km2. That density appears realistic because it is the one observed in the non hunted forest of the region. A second set of simulations uses the hunting data collected on the field.

Many scenarios have been implemented to test the influence of different hunting strategies: continuously repeating 1995 hunting data; removing traps from the spatial grid every 26 weeks and locating again at the same place or randomly re-locating within the same hunting location...

These experiments point out the crucial influence of the spatial dimension. We used CORMAS platform, (Common-pool resources and multiagent systems), a generic simulation environment based on Smalltalk which allows to build spatially-explicit individual-based models in a flexible way [11], developped in the GREEN research team of CIRAD [1].

We used multiagent approach as an alternatve to traditional mathematical tools: differential equations have been for a long time the adequate way to handle population dynamics. Multiagent systems offer efficient concepts to design, implement and simulate individual-based models. Nevertheless, simulations only provide short (finite) time information. In the field of natural resources management, it is of interest to know long term effects of managing decisions. The use of a formal approach may help to evaluate tendancies shown by simulation. Petri nets appear as the adequate approach which preserve formal analysis and individual-based modelling capabilities. More, efficient tools are available to design and analyze Petri net models.

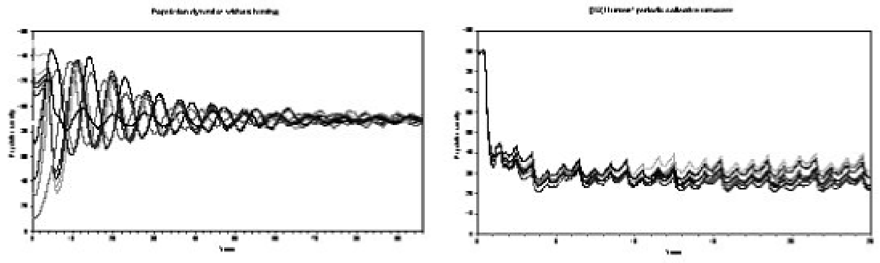


Fig. 1. Multiagent simulation results: the population size variation during time for two scenarios.

3 The Coloured Petri Net Model

Petri nets are a graphical and mathematical modelling tool applicable to many systems which the behavior can be described in terms of system states and their changes. Formally, a Petri net is a 5-tuple $PN = (P, T, F, W, M_0)$ where:

- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places,
- $T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
- $W : F \to \{1, 2, 3, ...\}$ is a weight function,
- $M_0 : P \to \{0, 1, 2, 3, ...\}$ is the initial marking,
- $P \cap T = \phi$ and $P \cup T \neq \phi$.

[1] http://cormas.cirad.fr/

A transition t is said to be enabled if each input place p of t is larked with at least $w(p,t)$ where $w(p,t)$ is the weight of the arc from p to t. A firing of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t,p)$ tokens to each output place p of t, where $w(t,p)$ is the weight of the arc from t to p. In graphical representation, places are drawn as circles, transitions as bars or boxes. Arcs are labeled with their weights (positive integers), where a $k-$weighted arc can be interpreted as the set of k parallel arcs. Labels for unity weight are usually omitted.

Coloured Petri nets allow the use of tokens that carry data value and can hence be distinguished one from the other. Arbitrary complex data types can be used as coloured sets, like a list of many thousand records, involving fields of many different types [12]. To be able to occur, a transition must have sufficient tokens in its input places, and this tokens must match the arc expression. We used CPN-AMI[2] which is a Petri Net based environment. It offers a set of services to ease the work of designers who specify systems and take benefits of the Petri net theory. It relies on the Macao graph Editor which also behaves as the User Interface of CPN-AMI [10].

3.1 Assumptions of the Model

The model is built by assembling modules which correspond each to an hypothesis. The main hypothesis are :

– H1: The moving hypothesis:

(a) The animals move randomly in the 4 directions (north, south, east and west); (b) The animals which move on an hunted cell are taken in the trap, according to a catch probability; (c) The animals taken in a trap are removed from the system (dead); (d) The maximum number of animals in a cell is defined; (e) The moves of the animals relate to all the population for each time step.

– H2: The population of animals increases by a fixed percentage
– H3: The growth hypothesis:

(a) The animals grow; (b) When an animal reach the fixed maximum age, it is removed from the system.

– H4: The schedule of the various actions is well defined.

Those assumptions have been grouped (as shown by their number) and a net module was built for each group. Modules share common places which will be used to connect each other. Each module contains two special non coloured and 1-bounded places: *Start_Hx* and *End_Hx* which are marked respectively at the beginnning and at the end of the "execution" of the module. The main places of the system are defined as follow:

[2] http://www.lip6.fr/cpn-ami

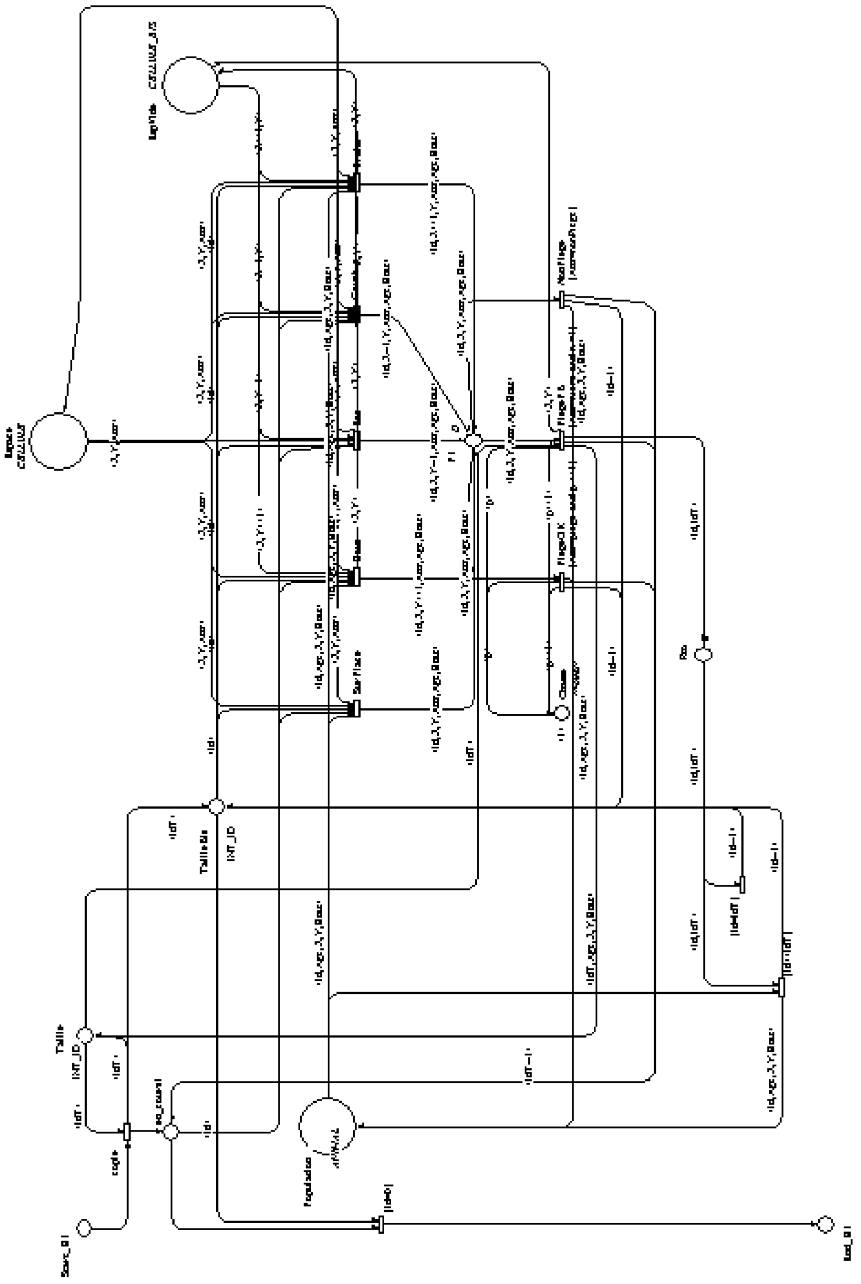


Fig. 2. The moving module

- A place named *Population* contains tokens $< Id, Age, X, Y, State >$. Each token represents an animal. *Age* is its age expressed in half-years, X and Y are the coordinates of its spatial position, Id its number and *State* represents the state of the animal.
- A place named *Space* contains tokens $< X, Y, Attr >$ corresponding each to a cell of the grid space similar to the space in the multiagent model. X and Y are the same as in the place named *Population. Attr* represents the state of the cell and express the fact that the cell is hunted or not, if it's water or road, or simply an empty cell.
- Some other places can be mentionned here: the place *Size* which contains a coloured token $< IdT >$ representing the population size; the place *SpaceEmpty* which contains tokens $< X, Y >$ for all the available positions in the space; the place *SpaceOcc* for all the occupied positions.

Below we have described two modules: the moving module corresponding to the first group of assumptions $H1$ which carry the spatial aspect of the model and the increasing population module corresponding to assumption $H2$.

3.2 The Moving Module

The module (figure 2) represents the moves of the whole population during a time step of the evolution of the system. Each animal can move to one of its four neighbour cells or stay on its cell.

When a token is put on place *Start_H*1, transition *copy* becomes enable and the population size $< IdT >$ is copied from place *Size* to place *SizeBis*, the place *step_*1 recieved a non coloured token. The token of the place *SizeBis* will be decreased during the moving process and will match the Id attribute of the token $< Id, Age, X, Y, State >$ currently moving. One of the transitions $Left, Right, Up$ or *Down* is firing according to the current moving direction. Otherwise, when there's no available position in the destination cell, the token stays in its initial position. After the moving step, the value of the token p and the state of the new position determine if the animal will be caught or not.

3.3 Increasing Population Module

This module (figure 3) increases the population size by creating a sub population which is added to the initial population. We have implemented in this module a divide mechanism which allows to increase the population according to the fixed increasing percentage defined by the model assumptions. The module starts by the marking of the place *Start_H*2 and ends when there is a token in the place *End_H*2. It shares places *SpaceOcc*, *Size*, *SizeBis* and *Population* with the moving module.

3.4 The Complete Modele

We have defined a scheduling module which connects End_H_i to $Start\_H_{i+1}$, and the last module to the first for repeating seasons. A control mechanism was

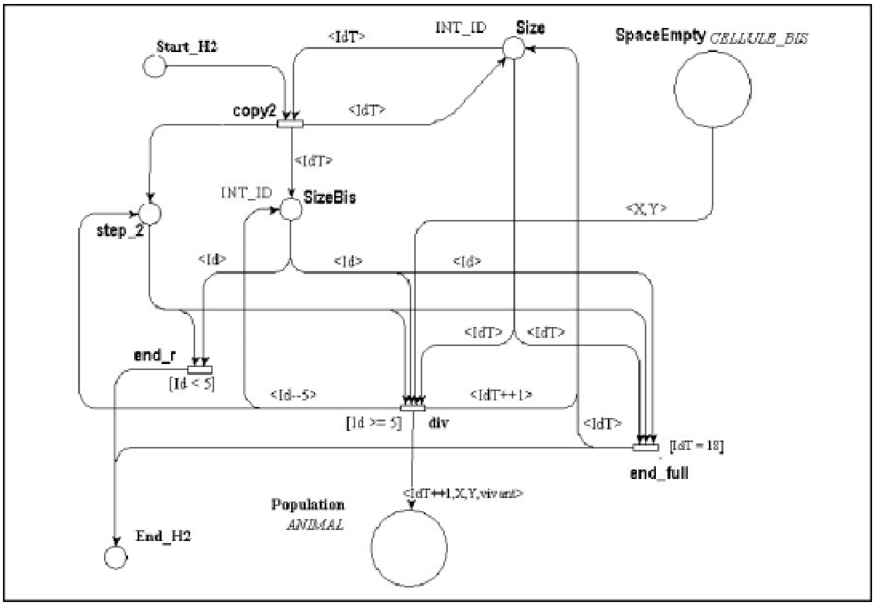


Fig. 3. Increasing population module

added to stop the system when the population size reaches zero value. The whole system was obtainaed in connecting modules by means of "channel places". It gave a Petri net with 20 places and 22 transitions.

4 Analysis of the Coloured Petri Net Model

We use simulations to validate each module of the model during the construction phase, by making sure if it had the expected behaviour on small initial marking. When the net is debugged, it can be analysed using structural properties analysis or model checking. The first way permits us to verify the coherence of the model. By computing invariants of places, we assume that the different control mechanisms we used in the model are well implemented.

The main phase of the analysis of the model is the study of its reachability graph. It is a directed graph with a node for each reachable system state and an arc for each possible transition from one system state to another. We used the PROD analysis tool which was integrated in CPN-AMI platform. Since the graph size is large, we experimented the model checker on smallest values of initial marking. We defined as initial marking a 2×2 space where each cell can contain at most two animals and where two of the four cells are hunted. With an initial population of 4 animals, we obtained a complete reachability graph with characteristics shown in the table 1.

We observed that the liveness of the system can be verified when the reachability graph doesn't have any terminal node. It's always the case when we alter-

Table 1. Statistics about the complete reachability graph of the Petri net model

configurations	permanent hunting seasons						alternating hunting and no hunting seasons (2)		
	maintaining trap positions (1a)			alternating trap positions (1b)					
	nodes	arcs	terminal nodes	nodes	arcs	terminal nodes	nodes	arcs	terminal nodes
◨	389584	412081	2	776396	820866	12	565206	583922	0
▚	443260	469479	6	884760	936715	0	724103	748391	0

nated hunting and non hunting seasons (2). For the scenario (1b), we obtain a significant difference betwen the two configurations. It appears as a consequence of the neighbourhood mode we implemented. When traps are always maintained on their position (1a), it is possible to reach a terminal node. These results show that there is an important correlation between spatial and temporal dynamics of the system.

5 Discussion

As explained by Singh & al. [13], in general, formalizations of agents systems have been used for two quite distinct purposes: as internal specification languages to be used by the agent in their reasoning or action and as external metalanguages to be used by the designers to specify, design and verify certain behavioral properties of agents situated in a dynamic environment. Our work can be classified under the second purpose. We are going to discuss here two aspects of the experiment: modelling and verification of properties.

The use of Coloured Petri nets to represent spatial behaviors induces to develop specific mechanisms to model certain dynamics:

- the choice of a moving direction;
- what happens if there is no empty position to reach;
- how to assume coherence within the different places of the model (*SpaceOcc* and *SpaceEmpty*, *Population* and *Size*).

The representation of random events too is not a trivial exercise. These difficulties are enough to divert the modeller's attention from the real problem. Furthermore, modelling a complex problem such as wildlife management using Petri nets enables us to get benefit from the formal semantic of the approach, and then avoid any ambiguity in the interpretation of the system description. System modularity allows maintenance facilities and avoid spaghetti effects which would appear if trying to design directly the complete net.

Model checking is a useful method for verifying properties of finite states systems. After formalizing a multiagent model, there is a real methodological challenge to achieve the verification part of the study, because of the system complexity. In our case, we are trying to know if they are (or are not) system states for which the population size has a nil value. We didn't obtain any significant difference when checking the model with that specific property or building the whole reachability graph. This is why we analyzed statistics about the complete reachability graph. It is clear that, due to the state-explosion problem, it would not be possible to analyze the model with realistic values. The model checker exhibits very interesting qualitative results, according to the field study. Formal analysis show the crucial influence of spatial dynamics (neighborhood's type, moving directions, trap's configuration) when studying the liveness of the system. These results have been pointed out by multiagent simulations.

6 Conclusion

The aim of this study was to validate simulation results we obtained from a multiagent model of hunting activity, using formal methods. We have built a coloured Petri net model with some assumptions taken from the multiagent model, and we analysed the reachability graph for some particular initial conditions which allow to prove the liveness of the system. In terms of management of natural resources, the liveness of our Petri net model is equivalent to the viability of the resources. We focussed our study on the qualitative aspect of the system and we showed correlations between the viability of the system, the spatial behaviour of animals and hunters spatio-temporal strategies. The use of coloured Petri nets, because of its call for precision, allows us to apprehend a very large specification. It gives us a good small-scale model to formally test the viability of some management strategies of the hunting activity and to validate simulation results of the multiagent model.

Acknowledgement. This work has been supported by a CNRS grant through the project "Systèmes Ecologiques et Action de l'Homme" and a CIRAD grant through the action "Validité des systèmes multiagents et Aide à la décision"

References

1. Barreteau O., Bousquet F.: *SHADOC: A Multi-Agent Model to Tackle Viability of Irrigated Systems.* Annals of Operations Research, 1998.
2. Bah A., Canal R., Aquino (D') P. Bousquet F.: *Les Systèmes Multi-Agents Génétiques Pour L'étude de la Mobilité Pastorale En Zone Intertropicale Sèche.* Actes du colloque SMAGET, 1998.
3. Rouchier J., Requier D. M.: *Un Travail Interdiscipliniare : Élaboration D'un Modèle de la Transhumace À L'extrême Nord Du Cameroun.* Actes du colloque SMAGET, 1998.
4. El Fallah S.: *Représentation et Manipulation de Plans À L'aide Des Réseaux de Petri.* Actes des 2èmes Journées Francophones IAD-SMA, Mai 1994.
5. Magnin L., Ferber J.: *Conception de Systèmes Multi-Agents Par Composants Modulaires et Réseaux de Petri.* Actes des Journées du PRC-IA, pp 129–140, 1993.

6. Gronewold Anja, Michael Sonnenschein: *Event-Based Modelling of Ecological Systems with Asynchronous Cellular Automata.* Ecological Modelling, 108:37–52, 1998.
7. Dubost G.: *L'écologie et la Vie Sociale Du Céphalophe Bleu (Cephalophus Monticola Thunberg), Petit Ruminant Forestier Africain.* Journal of Comparative Ethology, 1980.
8. Ferber J.: *Multiagent Systems: An Introduction to Distributed Artificial Intelligence* . Addison-Wesley 1999.
9. Bousquet F., Le Page C., Bakam I, Takforyan A.: *A Spatially Explicit Individual Based Model of Blue Duikers Population Dynamics: Multiagent Simulations of Bushmeat Hunting in an Eastern Cameroonian Village.* Accepted in Ecological Modelling, 1999.
10. Kordon F., Paviot-Adet E.: *Using CPN-AMI to Validate a Safe Channel Protocol.* The toolset Proceedings of the International Conference on Theory and Applications of Petri nets, Williamsburg, USA June 21-25 1999.
11. Bousquet F., Bakam I., Proton H., Le Page C.: *Cormas : Common-Pool Resources and Multi-Agents Systems.* Lectures Notes of Artificial Intelligence 1416, 2:826–837, 1998.
12. Jensen K.: *Coloured Petri Nets : Basic Concepts, Analysis Methods and Pratical Use* . Springer, 1997.
13. Singh Munidar, Anand S. Rao, Michael P. Georgeff: *Formal Methods in DAI: Logic Based Representation and Reasoning.* Multiagent Systems: A Modern Approach To Distributed Artificial Intelligence, 1999.

Formal Agent Development: Framework to System

Mark d'Inverno[1] and Michael Luck[2]

[1] Cavendish School of Computer Science, University of Westminster, London, UK
dinverm@westminster.ac.uk

[2] Department of Electronics and Computer Science, University of Southampton, UK
mml@ecs.soton.ac.uk

Abstract. Much work in the field of agent-based systems has tended to focus on either the development of practical applications of agent systems on the one hand, or the development of sophisticated logics for reasoning about agent systems on the other. Our own view is that work on formal models of agent-based systems are valuable inasmuch as they contribute to a fundamental goal of computing of practical agent development. In an ongoing project that has been running for several years, we have sought to do exactly that through the development of a formal framework that provides a conceptual infrastructure for the analysis and modelling of agents and multi-agent systems on the one hand, and enables implemented and deployed systems to be evaluated and compared on the other. In this paper, we describe our research programme, review its achievements to date, and suggest directions for the future.

1 Introduction

Over the course of the last ten years or so, work in agent-based systems has made dramatic progress. Indeed, there is now a relatively coherent and recognisable field of research that has experienced remarkable growth, both in the quantity of effort devoted by individual researchers and groups, and in the areas within it that have been subject to these investigations. Similarly, commercial development efforts have been increasing, with significant resources targetted at the area. It is a natural consequence of this rapid rise that there should be a range of approaches applied to the field, including both formal and practical (or empirical) methods. Yet in the rush to contribute to its development, the field of agent-based systems has experienced a fragmentation along the lines of a theoretical-practical divide, by which these two semi-distinct threads of research are advancing in parallel rather than interacting constructively.

As has been discussed elsewhere [12], much work has tended to focus on either the development of practical applications of agent systems on the one hand, or the development of sophisticated logics for reasoning about agent systems on the other. Certainly, both of these strands of research are important, but it is crucial for there to be a significant area of overlap between them for cross-fertilisation and for one strand to inform the other. Unfortunately, however, there has been a sizable gap between these formal models and implemented systems.

Our own view is that work on formal models of agent-based systems are valuable inasmuch as they contribute to a fundamental goal of computing of building real agent systems. This is not to trivialise or denigrate the effort of formal approaches, but to direct it towards integration in a broader research programme. In an ongoing project that has been running for several years, we have sought to do exactly that through the development of a formal framework that provides a conceptual infrastructure for the analysis and modelling of agents and multi-agent systems on the one hand, and enables implemented and deployed systems to be evaluated and compared on the other. In this paper, we describe our research programme, review its achievements to date, and suggest directions for the future. After briefly considering some related work on marrying the formal and practical, we review our earlier work on a formal agent framework and show how it leads to a detailed map of agent relationships. Then we examine plans in more detail, first at an abstract level, and then adding increasingly more detail to arrive at a specification of plans in dMARS. The paper ends by considering the value of this work in moving from abstract agent specification to detailed system description.

2 Background

Though the fragmentation into theoretical and practical aspects has been noted, and several efforts made in attempting to address this fragmentation in related areas of agent-oriented systems by, for example, Goodwin [11], Luck et al. [16], and Wooldridge and Jennings [21], much remains to be done in bringing together the two strands of work.

This section draws on Luck's outline [12] of the ways in which some progress has been made with BDI agents [2], a well-known and effective agent architecture. Rao, in particular, has attempted to unite BDI theory and practice in two ways. First, he provided an abstract agent architecture that serves as an idealization of an implemented system and as a means for investigating theoretical properties [19]. Second, he took an alternative approach by starting with an implemented system and then formalizing the operational semantics in an agent language, AgentSpeak(L), which can be viewed as an abstraction of the implemented system, and which allows agent programs to be written and interpreted [18].

In contrast to this approach, some work aims at constructing directly executable formal models. For example, Fisher's work on Concurrent MetateM [9] has attempted to use temporal logic to represent individual agent behaviours where the representations can either be executed directly, verified with respect to a logical requirement, or transformed into a more refined representation. Further work aims to use this to produce a full development framework from a single high-level agent to a cooperating multi-agent system. In a similar vein, Parsons et al. [17] aim to address the gap between specification and implementation of agent architectures by viewing an agent as a multi-context system in which each architectural component is represented as a separate unit, an encapsulated set of axioms, and an associated deductive mechanism whose interrelationships are specified using bridge rules. Since theorem-provers already exist for multi-context systems, agents specified in this way can also be directly excecuted.

As yet, the body of work aimed at bridging the gap between theory and practice is small. Fortunately, though, there seems to be a general recognition that one of the key roles of theoretical and practical work is to inform the other [4], and while this is made difficult by the almost breakneck pace of progress in the agent field, that recognition bodes well for the future. Some sceptics remain, however, such as Nwana, who followed Russell in warning against *premature mathematization*, and the danger that lies in wait for agent research [1].

3 Overview

As stated above, we view our enterprise as that of building programs. In order to do so, however, we need to consider issues at different points along what we call the *agent development line*, identifying the various foci of research in agent-based systems in support of final deployment, as shown in Figure 1. To date, our work has concentrated on the first three of the stages identified.

– We have provided a formal agent framework within which we can explore some fundamental questions relating to agent architectures, configurations of multi-agent systems, inter-agent relationships, and so on, independent of any particular model. The framework continues to be extended to cover a broader range of issues, and to provide a more complete and coherent conceptual infrastructure.
– In contrast to starting with an abstract framework and refining it down to particular system implementations, we have also attempted to start with specific deployed systems and provide formal analyses of them. In this way, we seek to move backwards to link the system specifications to the conceptual formal framework, and also to provide a means of comparing and evaluating competing agent systems.
– The third strand aims to investigate the process of moving from the abstract to the concrete, through the construction of agent development methodology, an area that has begun to receive increasing attention. In this way, we hope to marry the value of formal analysis with the imperative of systems development in a coherent fashion, leading naturally to the final stage of the development line, to *agent deployment*.

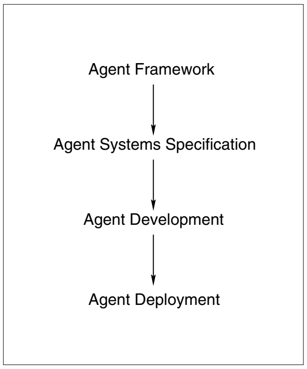


Fig. 1. The Agent Development Line

This paper can be seen as a continuation of the work contained in [7], which describes the research programme at an earlier stage of development. That work introduced requirements for formal frameworks, and showed how our agent framework satisfied those requirements in relation to, in particular, goals generation and adoption, some initial inter-agent relationships, and their application to the Contract Net protocol. In this paper, we build on that work, showing further levels of analysis of agent relationships, and also describe further work on formal agent specification.

In what follows, we use the Z specification language [20], for reasons of accessibility, clarity and existing use in software development. The arguments are well-rehearsed and can be found in numerous of the references. Similarly, we assume familiarity with the notation, details of which can also be found in many of the references, and especially in [20]. The specification in this paper is not intended to be complete, nor to provide the most coherent exposition of a particular piece of work, but to show how a broad research programme in support of the aims above is progressing. Details of the different threads of work may be found in the references in each of the relevant sections.

4 The Formal Agent Framework

We begin by briefly reviewing earlier work. In short, we propose a four-tiered hierarchy comprising *entities*, *objects*, *agents* and *autonomous agents* [13]. The basic idea underlying this hierarchy is that all components of the world are entities. Of these entities, some are objects, of which some, in turn, are agents and of these, some are autonomous agents, as shown in Figure 2.

$$
\begin{array}{|l}
\hline
\textit{Entity} \\
\textit{attributes} : \mathbb{P}\,\textit{Attribute} \\
\textit{capableof} : \mathbb{P}\,\textit{Action} \\
\textit{goals} : \mathbb{P}\,\textit{Goal} \\
\textit{motivations} : \mathbb{P}\,\textit{Motivation} \\
\hline
\textit{attributes} \neq \{\ \} \\
\hline
\end{array}
$$

$$
\begin{aligned}
&\textit{Object} == [\textit{Entity} \mid \textit{capableof} \neq \{\ \}] \\
&\textit{Agent} == [\textit{Object} \mid \textit{goals} \neq \{\ \}] \\
&\textit{AutoAgent} == [\textit{Agent} \mid \textit{motivations} \neq \{\ \}] \\
&\textit{NeutralObject} == [\textit{Object} \mid \textit{goals} = \{\}] \\
&\textit{ServerAgent} == [\textit{Agent} \mid \textit{motivations} = \{\}]
\end{aligned}
$$

Fig. 2. Entities in the Agent Framework

Entities serve as a useful abstraction mechanism by which they are regarded as distinct from the remainder of the environment, to organise perception. An object is then an entity with abilities which can affect environments in which it is situated. An agent is just an object either that is useful to another agent where this usefulness is defined in terms of satisfying that agent's goals, or that exhibits independent purposeful behaviour. In other words, an agent is an object with an associated set of goals. This definition of agents relies upon the existence of such other agents which provide the goals that are adopted instantiate an agent. In order to escape an infinite regress of goal adoption, we define autonomous agents, which are agents that generate their own goals from motivations. We also distinguish those objects that are not agents, and those agents that are not autonomous and refer to them as *neutral-objects* and *server-agents* respectively. An agent is then either a server-agent or an autonomous agent, and an object is either a neutral-object or an agent.

5 Inter-agent Relationships

Agents and autonomous agents are thus defined in terms of goals. Agents *satisfy* goals, while autonomous agents may, additionally, generate them. Goals may be adopted by either autonomous agents, non-autonomous agents or objects without goals. Since non-autonomous agents satisfy goals for others they *rely* on other agents for purposeful existence, indicating that goal adoption creates critical inter-agent relationships. The combined total of these agent relationships defines a social organisation that is not artificially or externally imposed but arises as a natural and elegant consequence of our definitions of agents and autonomous agents. Thus the agent framework allows an explicit and precise analysis of multi-agent systems with no more conceptual primitives than were introduced for the initial framework to describe individual agents.

DirectEngagement

$client : Agent;\ server : ServerAgent$
$goal : Goal$

$client \neq server$
$goal \in (client.goals \cap server.goals)$

Cooperation

$goal : Goal;\ generatingagent : AutoAgent$
$cooperatingagents : \mathbb{P}\, AutoAgent$

$goal \in generatingagent.goals$
$\forall\, aa : cooperatingagents \bullet goal \in aa.goals$
$generatingagent \notin cooperatingagents$
$cooperatingagents \neq \{\,\}$

Fig. 3. Engagement and Cooperation

To illustrate, let us consider two particular relationships, direct engagements and cooperations [14]. In a direct engagement, a *client*-agent with some goals uses another *server*-agent to assist them in the achievement of those goals. A server-agent either exists already as a result of some other engagement, or is instantiated from a neutral-object for the current engagement. (No restriction is placed on a client-agent.) We define a *direct engagement* in Figure 3 to consist of a client agent, *client*, a server agent, *server*, and the goal that *server* is satisfying for *client*. An agent cannot engage itself, and both agents must have the goal of the engagement. By contrast, two autonomous agents are said to be *cooperating* with respect to some goal if one of the agents has adopted goals of the other. Using these fundamental forms of interaction, we can proceed to define a more detailed taxonomy of inter-agent relationships that allows a richer understanding of the social configuration of agents, suggesting different possibilities for interaction, as shown in Figure 4, taken from [15]. Importantly, the relationships identified are not imposed on multi-agent systems, but arise naturally from agents interacting, and therefore underlie all multi-agent systems.

directly engages	$\forall\, c : Agent;\ s : ServerAgent \bullet (c, s) \in dengages \Leftrightarrow$ $\exists\, d : dengagements \bullet d.client = c \wedge d.server = s$
engages	$\forall\, c : Agent, s : ServerAgent \bullet (c, s) \in engages \Leftrightarrow$ $\exists\, ec : engchains \bullet (s \in (\operatorname{ran} ec.agentchain) \wedge c = ec.autoagent) \vee$ $(((c, s), ec.agentchain) \in before)$
indirectly engages	$indengages = engages \setminus dengages$
owns	$\forall\, c : Agent;\ s : ServerAgent \bullet (c, s) \in owns \Leftrightarrow$ $(\forall\, ec : engchains \mid s \in \operatorname{ran} ec.agentchain \bullet$ $ec.autoagent = c\ \vee\ ((c, s), ec.agentchain) \in before)$
directly owns	$downs = owns \cap dengages$
uniquely owns	$\forall\, c : Agent;\ s : ServerAgent \bullet (c, s) \in uowns \Leftrightarrow$ $(c, s) \in\ downs \wedge \neg\, (\exists\, a : Agent \mid a \neq c \bullet (a, s) \in engages)$
specifically owns	$\forall\, c : Agent;\ s : ServerAgent \bullet (c, s) \in sowns \Leftrightarrow$ $(c, s) \in\ owns \wedge \#(s.goals) = 1$

Fig. 4. Inter-agent Relationships

The direct engagement relationship specifies the situation in which there is a direct engagement for which the first agent is the client and the second agent is the server. In general, however, any agent involved in an engagement chain engages all those agents that appear subsequently in the chain. To distinguish engagements involving an intermediate agent we introduce the indirect engagement relation *indengages*; an agent *indirectly* engages another if it engages it, but does not *directly* engage it. If many agents directly engage the same entity, then no single agent has complete control over it. It is important to understand *when* the behaviour of an engaged entity can be modified without any deleterious effect (such as when no other agent uses the entity for a *different* purpose). In this case we say that the agent *owns* the entity. An agent, c, owns another agent, s, if, for every sequence of server-agents in an engagement chain, ec, in which s appears, c precedes it, or c is the autonomous client-agent that initiated the chain. An agent *directly owns* another if it owns it and directly engages it. We can further distinguish the *uniquely owns* relation, which holds when an agent *directly* and *solely* owns another, and *specifically owns*, which holds when it owns it, and has only one goal.

6 Sociological Behaviour

Social behaviour involves an agent interacting with others; sociological behaviour requires more, that an agent understand its relationships with others. In order to do so, it must model them, their relationships, and their plans.

6.1 Models and Plans

To model their environment, agents require an *internal store*, without which their past experience could not direct behaviour, resulting in reflexive action alone. A store exists as part of an agent's state in an environment but it must also have existed *prior* to that state. We call this feature an *internal store* or *memory*, and define *store agents* as those with memories. Unlike *social* agents that engage in interaction with others, *sociological* agents model relationships as well as agents. It is a simple matter to define the model an agent has of another agent (*AgentModel*), by re-using the agent framework components as shown in Figure 5. Even though the types of these constructs are equivalent to those presented earlier, it is useful to distinguish physical constructs from mental constructs such as models, as it provides a conceptual aid. We can similarly define models of other components and relationships so that specifying a sociological agent amounts to a refinement of the *Agent* schema as outlined in Figure 5.

Now, in order to consider sociological agents, or agents that can model others, we can construct a high-level model of *plan-agents* that applies equally well to reactive or deliberative, single-agent or multi-agent, planners. It represents a high-level of abstraction without committing to the nature of an agent, the plan representation, or of the agent's environment; we simply distinguish *categories* of plan and possible relationships between an agent's plans and goals. Specifically, we define *active* plans as those identified as candidate plans not yet selected for execution; and *executable* plans as those active plans that have been selected for execution.

Formally, we initially define the set of all agent plans to be a given set ([*Plan*]), so that at this stage we abstract out any information about the nature of plans themselves. Our highest-level description of a *plan-agent* can then be formalised in the *PlanAgent* schema of Figure 6.

The variables *goallibrary*, *planlibrary*, *activeplans* and *executableplans* represent the agent's repository of goals, repository of plans, active plans and executable plans,

$$AgentModel == Agent$$
$$CoopModel == Cooperation$$
$$\vdots$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ SociologicalAgent \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Agent \\
agentmodels : \mathbb{P}\, AgentModel \\
coopmodels : \mathbb{P}\, CoopModel \\
\vdots \\
\hline
\end{array}$$

Fig. 5. Sociological Agent

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ PlanAgent \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Agent \\
goallibrary : \mathbb{P}\, Goal \\
planlibrary, activeplans, executableplans : \mathbb{P}\, Plan \\
activeplangoal, plangoallibrary : Goal \nrightarrow \mathbb{P}\, Plan \\
\hline
\operatorname{dom} activeplangoal \subseteq goals \;\wedge\; \bigcup(\operatorname{ran} activeplangoal) = activeplans \\
\operatorname{dom} plangoallibrary \subseteq goallibrary \;\wedge\; \bigcup(\operatorname{ran} plangoallibrary) \subseteq planlibrary \\
goals \subseteq goallibrary \;\wedge\; executableplans \subseteq activeplans \subseteq planlibrary \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ SPAgent \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
SociologicalAgent \\
PlanAgent \\
\hline
\end{array}$$

Fig. 6. Sociological Plan Agent

respectively. Each active plan is necessarily associated with one or more of the agent's current goals as specified by *activeplangoal*. For example, if the function contains the pair $(g, \{p_1, p_2, p_3\})$, it indicates that p_1, p_2 and p_3 are competing active plans for g. Whilst active plans must be associated with at least one active goal the converse is not true, since agents may have goals for which no plans have been considered. Analogously the *plangoallibrary* function relates the repository of goals, *goallibrary*, to the repository of plans, *planlibrary*. However, not necessarily all library plans and goals are related by this function.

6.2 Plan and Agent Categories

Now, using these notions, we can describe some example categories of goals, agents and plans (with respect to the models of the sociological plan-agent), that may be relevant to an agent's understanding of its environment. Each of the categories is formally defined in Figure 7, in which the sociological plan-agent is denoted as *spa*. Any variable preceded by *model* denotes the models that *spa* has of some specific type of entity or relationship. For example, *spa.modelneutralobjects* and *spa.modelowns* are the neutral objects and ownership relations the sociological agent models.

A *self-sufficient plan* is any plan that involves only neutral-objects, server-agents the plan-agent owns, and the plan-agent itself. Self-sufficient plans can therefore be executed without regard to other agents, and exploit current agent relationships. (The formal definition makes use of the relational image operator: in general, the relational image $R(\!| \; S \; |\!)$ of a set S through a relation R is the set of all objects y to which R relates to some member x of S.) A *self-sufficient goal* is any goal in the goal library that has an associated self-sufficient plan. These goals can then, according to the agent's model, be achieved independently of the existing social configuration. A *reliant-goal* is any goal that has a non-empty set of associated plans that is not self-sufficient.

self-suff plan

$$\forall p \in spa.planlibrary \bullet selfsuff(p) \Leftrightarrow spa.planentities(p) \subseteq \\ spa.modelneutralobjects \cup spa.modelself \cup spa.modelowns(\!| \; spa.modelself \; |\!) \bullet p$$

self-suff goal

$$\forall g \in spa.goallibrary \bullet selfsuffgoal(g) \Leftrightarrow (\exists p \in spa.plangoallibrary(g) \bullet p \in selfsuff)$$

reliant goal

$$\forall g \in spa.goallibrary \bullet reliantgoal(g) \Leftrightarrow spa.plangoallibrary\ g \neq \{\ \} \ \wedge \\ \neg\ (\exists p : spa.plangoallibrary\ g \bullet p \in selfsuff)$$

Fig. 7. Sociological Agent Categories I

For each plan that is not self-sufficient, a sociological plan-agent can establish the autonomous agents that may be affected by its execution, which is an important criterion in selecting a plan from competing active plans. An autonomous agent *A* may be affected by a plan in one of two ways: either it is required to perform an action directly, or it is engaging a server-agent *S* required by the plan. In this latter case, a sociological plan-agent can reason about either persuading *A* to share or release *S*, taking *S* without permission, or finding an alternative server-agent or plan. To facilitate such an analysis, we can define further categories of agents and plans, as described in [15],but we do not consider them further here.

7 Agent Plans

Now, the rather abstract notion of plans above can be refined further, and we can develop a more detailed specification that will be useful when considering different forms of agents and planning abilities. We begin by introducing some general concepts, and then move to a detailed description of plans in dMARS.

7.1 Modelling Plans

This involves defining first the *components* of a plan, and then the *structure* of a plan, as shown in Figure 8. The components, which we call *plan-actions*, each consist of a *composite-action* and a set of related entities as described below. The structure of plans defines the relationship of the component plan-actions to one another. For example, plans may be *total* and define a sequence of plan-actions, *partial* and place a partial order on the performance of plan-actions, or *trees* and, for example, allow choice between alternative plan-actions at every stage in the plan's execution.

$$\begin{aligned}
&\mathit{Primitive} == \mathit{Action}\\
&\mathit{Template} == \mathbb{P}\,\mathit{Action}\\
&\mathit{ConcPrimitive} == \mathbb{P}\,\mathit{Action}\\
&\mathit{ConcTemplate} == \mathbb{P}(\mathbb{P}\,\mathit{Action})\\
&\mathit{ActnComp} ::= \mathit{Prim}\langle\!\langle \mathit{Primitive}\rangle\!\rangle\\
&\quad \mid \mathit{Temp}\langle\!\langle \mathit{Template}\rangle\!\rangle\\
&\quad \mid \mathit{ConcPrim}\langle\!\langle \mathit{ConcPrimitive}\rangle\!\rangle\\
&\quad \mid \mathit{ConcTemp}\langle\!\langle \mathit{ConcTemplate}\rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{TotalPlan} &== \operatorname{seq} \mathit{PlanAction}\\
\mathit{TreePlan} &::= \mathit{Tip}\langle\!\langle \mathit{PlanAction}\rangle\!\rangle\\
&\quad\mid \mathit{Fork}\langle\!\langle \mathbb{P}_1(\mathit{PlanAction} \times \mathit{TreePlan})\rangle\!\rangle\\
\mathit{Plan} &::= \mathit{Part}\langle\!\langle \mathit{PartialPlan}\rangle\!\rangle\\
&\quad\mid \mathit{Total}\langle\!\langle \mathit{TotalPlan}\rangle\!\rangle\\
&\quad\mid \mathit{Tree}\langle\!\langle \mathit{TreePlan}\rangle\!\rangle
\end{aligned}$$

$$\begin{array}{|l}
\mathit{planpairs} : \mathit{Plan} \to \mathbb{P}\,\mathit{PlanAction}\\
\mathit{planentities} : \mathit{Plan} \to \mathbb{P}\,\mathit{EntityModel}\\
\mathit{planactions} : \mathit{Plan} \to \mathbb{P}\,\mathit{Action}
\end{array}$$

$$\begin{aligned}
&\mathit{PlanAction} == \mathbb{P}(\mathit{ActnComp} \times \mathbb{P}\,\mathit{EntityModel})\\
&\mathit{PartialPlan} == \{ps : \mathit{PlanAction} \leftrightarrow \mathit{PlanAction} \mid \forall\, a, b : \mathit{PlanAction} \bullet\\
&\qquad (a,a) \notin ps^{+} \land (a,b) \in ps^{+} \Rightarrow (b,a) \notin ps^{+} \bullet ps\}
\end{aligned}$$

Fig. 8. Plan components and structure

We identify four types of action that may be contained in plans, called *primitive*, *template*, *concurrent-primitive* and *concurrent-template*. There may be other categories and variations on those we have chosen, but not only do they provide a starting point for specifying systems, they also illustrate how different representations can be formalised and incorporated within the same model. A primitive action is simply a base action as defined in the agent framework, and an action template provides a high-level description of what is required by an action, defined as the set of all primitive actions that may result through an instantiation of that action-template. An example where the distinction is manifest is in dMARS (see Section 7.2), where template actions would represent action formulae containing free variables. Once all the free variables are bound to values, the action is then a primitive action and can be performed. We also define a concurrent-primitive action as a set of primitive actions to be performed concurrently and a concurrent action-template as a set of template actions that are performed concurrently. A new type, *ActnComp*, is then defined as a *compound-action* to include all four of these types.

Actions must be performed by entities, so we associate every composite-action in a plan with a set of entities, such that each entity in the set can potentially perform the action. At some stage in the planning process this set may be empty, indicating that no choice of entity has yet been made. We define a *plan-action* as a set of pairs, where each pair contains a composite-action and a set of those entities that could potentially perform the action. Plan-actions are defined as a *set of pairs* rather than a *single pair* so that plans containing simultaneous actions can be represented.

We specify three commonly-found categories of plan according to their structure as discussed earlier, though other types may be specified similarly. A partial plan imposes a partial order on the execution of actions, subject to two constraints. First, an action cannot be performed before itself and, second, if plan-action a is before b, b cannot be before a. A plan consisting of a total order of plan-actions is a total plan. Formally, this is represented as a sequence of plan-actions. A plan that allows a choice between actions at every stage is a tree. In general, a tree is either a leaf node containing a plan-action, or a fork containing a node, and a (non-empty) set of branches each leading to a tree. These are formalised in Figure 8, replacing the definition of *Plan* as a given set by a free-type definition to include the three plan categories thus defined.

7.2 Application to BDI Agents

While many different and contrasting single-agent architectures have been proposed, perhaps the most successful are those based on the belief-desire-intention (BDI) framework. In particular, the Procedural Reasoning System (PRS), orginally described in 1987 [10], has progressed from an experimental LISP version to a full C++ implementation known as the distributed Multi-Agent Reasoning System (dMARS). It has been applied in perhaps the most significant multi-agent applications to date. As described in Section 2 above, PRS, which has its conceptual roots in the belief-desire-intention (BDI) model of practical reasoning developed by Bratman [2], has been the subject of a dual approach by which a significant commercial system has been produced while the theoretical foundations of the BDI model continue to be closely investigated.

As part of our work, we have sought to formalise these BDI systems through the direct representation of the implementations on the one hand, and through refinement of the detailed models constructed through the abstract agent framework on the other. This work has included the formal specification [8] of the AgentSpeak(L) language developed by Rao [18], which is a programming language based on an abstraction of the PRS architecture; irrelevant implementation detail is removed, and PRS is stripped to its bare essentials. Our specification reformalises Rao's original description so that it is couched in terms of state and operations on state that can be easily refined into an implemented system. In addition, being based on a simplified version of dMARS, the specification provides a starting point for actual specifications of these more sophisticated systems. Subsequent work continued this theme by moving to produce an abstract formal specification of dMARS itself, through which an operational semantics for dMARS was provided, offering a benchmark against which future BDI systems and PRS-like implementations can be compared.

Due to space constraints, we cannot hope to get anywhere near a specification of either of these systems, but instead we aim to show how we can further refine the models of plans described above to get to a point at which we can specify the details of such implementations. The value of this is in the ease of comparison and analysis with the more abstract notions described earlier.

We begin our specification, shown in Figure 9 by defining the allowable *beliefs* of an agent in dMARS, which are like PROLOG facts. To start, we define a *term*, which is either a variable or a function symbol applied to a (possibly empty) sequence of terms, and an *atom*, a predicate symbol applied to a (possibly empty) sequence of terms. In turn, a *belief formula* is either an atom or the negation of an atom, and the set of *beliefs* of an agent is the set of all ground belief formulae (i.e. those containing no variables). (We assume an auxiliary function *belvars* which, given a belief formula, returns the set of variables it contains.) Similarly, a situation formula is an expression whose truth can be evaluated with respect to a set of beliefs. A goal is then a belief formula prefixed with an achieve operator or a situation formula prefixed with a query operator. Thus an agent can have a goal either of achieving a state of affairs or of determining whether the state of affairs holds.

The types of action that agents can perform may be classified as either *external* (in which case the domain of the action is the environment outside the agent) or *internal* (in which case the domain of the action is the agent itself). External actions are specified as if they are procedure calls, and comprise an external action symbol (analogous to the procedure name) taken from the set [*ActionSym*], and a sequence of terms (analogous to the parameters of the procedure). Internal actions may be one of two types: add or remove a belief from the data base.

Plans are *adopted* by agents and, once adopted, constrain an agent's behaviour and act as *intentions*. They consists of six components: an *invocation condition* (or *triggering event*); an optional *context* (a situation formula) that defines the pre-conditions of the plan, i.e., what must be believed by the agent for a plan to be executable; the *plan body*, which is a tree representing a kind of flow-graph of actions to perform; a *maintenance condition* that must be true for the plan to continue executing; a set of *internal actions* that are performed if the plan succeeds; and finally, a set of *internal actions* that are performed if the plan fails. The tree representing the body has states as nodes, and arcs (branches) representing either a goal, an internal action or an external action as defined below. Executing a plan successfully involves traversing the tree from the root to any leaf node.

A trigger event causes a plan to be adopted, and four types of events are allowable as triggers: the acquisition of a new belief; the removal of a belief; the receipt of a message; or the acquisition of a new goal. This last type of trigger event allows goal-driven as well as event-driven processing. As noted above, plan bodies are trees in which arcs are labelled with either goals or actions and states are place holders. Since states are

$[\mathit{Var}, \mathit{FunSym}, \mathit{PredSym}]$

$\mathit{Term} ::= \mathit{var}\langle\!\langle \mathit{Var} \rangle\!\rangle \mid \mathit{functor}\langle\!\langle \mathit{FunSym} \times \mathrm{seq}\,\mathit{Term} \rangle\!\rangle$

Atom

$\mathit{head} : \mathit{PredSym}$

$\mathit{terms} : \mathrm{seq}\,\mathit{Term}$

$\mathit{BeliefFormula} ::= \mathit{pos}\langle\!\langle \mathit{Atom} \rangle\!\rangle \mid \mathit{not}\langle\!\langle \mathit{Atom} \rangle\!\rangle$

$\mathit{Belief} == \{b : \mathit{BeliefFormula} \mid \mathit{belvars}\ b = \varnothing \bullet b\}$

$$\begin{aligned} \mathit{SituationFormula} ::= {} & \mathit{belform}\langle\!\langle \mathit{BeliefFormula} \rangle\!\rangle \\ \mid {} & \mathit{and}\langle\!\langle \mathit{SituationFormula} \times \mathit{SituationFormula} \rangle\!\rangle \\ \mid {} & \mathit{or}\langle\!\langle \mathit{SituationFormula} \times \mathit{SituationFormula} \rangle\!\rangle \\ \mid {} & \mathit{true} \mid \mathit{false} \end{aligned}$$

$\mathit{Goal} ::= \mathit{achieve}\langle\!\langle \mathit{BeliefFormula} \rangle\!\rangle \mid \mathit{query}\langle\!\langle \mathit{SituationFormula} \rangle\!\rangle$

ExtAction

$\mathit{name} : \mathit{ActionSym}$

$\mathit{terms} : \mathrm{seq}\,\mathit{Term}$

$\mathit{IntAction} ::= \mathit{add}\langle\!\langle \mathit{BeliefFormula} \rangle\!\rangle \mid \mathit{remove}\langle\!\langle \mathit{BeliefFormula} \rangle\!\rangle$

$$\begin{aligned} \mathit{TriggerEvent} ::= {} & \mathit{addbelevent}\langle\!\langle \mathit{Belief} \rangle\!\rangle \\ \mid {} & \mathit{rembelevent}\langle\!\langle \mathit{Belief} \rangle\!\rangle \\ \mid {} & \mathit{toldevent}\langle\!\langle \mathit{Atom} \rangle\!\rangle \\ \mid {} & \mathit{goalevent}\langle\!\langle \mathit{Goal} \rangle\!\rangle \end{aligned}$$

$\mathit{Branch} ::= \mathit{extaction}\langle\!\langle \mathit{ExtAction} \rangle\!\rangle \mid \mathit{intaction}\langle\!\langle \mathit{IntAction} \rangle\!\rangle \mid \mathit{subgoal}\langle\!\langle \mathit{Goal} \rangle\!\rangle$

$\mathit{Body} ::= \mathit{End}\langle\!\langle \mathit{State} \rangle\!\rangle \mid \mathit{Fork}\langle\!\langle \mathbb{P}_1(\mathit{State} \times \mathit{Branch} \times \mathit{Body}) \rangle\!\rangle$

Plan

$\mathit{inv} : \mathit{TriggerEvent}$

$\mathit{context} : \mathit{optional}[\mathit{SituationFormula}]$

$\mathit{body} : \mathit{Body}$

$\mathit{maint} : \mathit{SituationFormula}$

$\mathit{succ} : \mathrm{seq}\,\mathit{IntAction}$

$\mathit{fail} : \mathrm{seq}\,\mathit{IntAction}$

Fig. 9. Plans in dMARS

not important in themselves, we define them using the given set [*State*]. An arc (branch) within a plan body may be labelled with either an internal or external action, or a subgoal. Finally, a dMARS plan body is either an *end tip* containing a state, or a *fork* containing a state and a non-empty set of branches each leading to another tree.

All these components can then be brought together into the definition of a plan. The basic execution mechanism for dMARS agents involves an agent matching the trigger and context of each plan against the chosen event in the event queue and the current set of beliefs, respectively, and then generating a set of candidate, matching plans, selecting one, and making a *plan instance* for it. Space constraints prohibit going into further details of the various aspects of this work, but we hope that it has been possible to show how increasing levels of analysis and detail enable transition between abstract conceptual infrastructure and implemented system.

8 Summary

Our efforts with BDI agents [6,5] have provided formal computational models of implemented systems and idealised systems, using the Z specification language [20], a standard (and commonly-used) formal method of software engineering. The benefit of this work is that the formal model is much more strongly related to the implementation, in that it can be checked for type-correctness, it can be animated to provide a prototype system, and it can be formally and systematically refined to produce a provably correct implementation. In this vein, related work has sought to contribute to the conceptual and theoretical foundations of agent-based systems through the use of such specification languages (used in traditional software engineering) that enable formal modelling yet provide a basis for implementation of practical systems, as has been done by several researchers [3,11,8].

Indeed, as the fields of intelligent agents and multi-agent systems move relentlessly forwards, it is becoming increasingly more important to maintain a coherent world view that both structures existing work and provides a base on which to keep pace with the latest advances. Our framework has allowed us to do just that. By elaborating the agent hierarchy in different ways, we have been able to detail both individual agent functionality and develop models of evolving social relationships between agents with, for example, our analyses of goal generation and adoption, and our treatment of engagement and cooperation. Not only does this provide a clear conceptual foundation, it also allows us to refine our level of description to particular systems and theories.

Acknowledgement. Thanks to Simon Miles and Kevin Bryson for working through this paper and identifying some errors at very short notice.

References

1. R. Aylett, F. Brazier, N. Jennings, M. Luck, C. Preist, and H. Nwana. Agent systems and applications. *Knowledge Engineering Review*, 13(3):303–308, 1998.
2. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.
3. I. D. Craig. *The Formal Specification of Advanced AI Architectures*. Ellis Horwood, 1991.
4. M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge. Formalisms for multi-agent systems. *Knowledge Engineering Review*, 12(3):315–321, 1997.
5. M. d'Inverno, D. Kinny, and M. Luck. Interaction protocols in agentis. In *ICMAS'98, Third International Conference on Multi-Agent Systems*, pages 112–119, Paris, France, 1998. IEEE Computer Society.

6. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, volume 1365, pages 155–176. Springer-Verlag, 1998.
7. M. d'Inverno and M. Luck. Development and application of a formal agent framework. In M. G. Hinchey and L. Shaoying, editors, *ICFEM'97: First IEEE International Conference on Formal Engineering Methods*, pages 222–231. IEEE Press, 1997.
8. M. d'Inverno and M. Luck. A formal specification of agentspeak(l). *Logic and Computation*, 8(3), 1998.
9. M. Fisher. Representing and executing agent-based systems. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI 890)*, pages 307–323. Springer, 1995.
10. M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682. AAAI Press / MIT Press, 1987.
11. R. Goodwin. A formal specification of agent properties. *Journal of Logic and Computation*, 5(6):763–781, 1995.
12. M. Luck. From definition to deployment: What next for agent-based systems? *The Knowledge Engineering Review*, 14(2):119–124, 1999.
13. M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.
14. M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In *Proceedings of the First Australian DAI Workshop, Lecture Notes in Artificial Intelligence, 1087*, pages 70–84. Springer Verlag, 1996.
15. M. Luck and M. d'Inverno. Plan analysis for autonomous sociological agents. In *Submitted to the Seventh International Workshop on Agent Theories, Architectures and Languages*, 2000.
16. M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In *Intelligent Agents III, LNAI 1193*, pages 49–63. Springer, 1997.
17. S. Parsons, C. Sierra, and N. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.
18. A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038*, pages 42–55. Springer-Verlag, 1996.
19. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning*, pages 439–449, 1992.
20. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Hemel Hempstead, 2nd edition, 1992.
21. M. J. Wooldridge and N. R. Jennings. Formalizing the cooperative problem solving process. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, 1994.

Automatic Synthesis of Agent Designs in UML

Johann Schumann[1] and Jon Whittle[2]

[1] RIACS / NASA Ames, Moffett Field, CA 94035 USA,
schumann@ptolemy.arc.nasa.gov
[2] QSS / NASA Ames, Moffett Field, CA 94035 USA,
jonathw@ptolemy.arc.nasa.gov

Abstract. It is anticipated that the UML, perhaps with domain-specific extensions, will increasingly be used to model and analyse agent-based systems. Current commercial tools for UML, however, contain a number of gaps that limit this growth potential. As an example, there is little or no support for automatic translations between UML notations. We present one such translation — from sequence diagrams to statecharts — and discuss how such an algorithm could be used in agent modeling. In contrast to other approaches, our algorithm makes a justified merging of the input sequence diagrams based on simple logical specifications of messages passed between agents/objects, and detects conflicting behaviors in different sequence diagrams. In addition, we generate statecharts that make full use of hierarchy, leading to generated designs that more closely resemble those which a designer might produce. This is important in the context of iterative design, since the designer will likely want to modify the generated statecharts to refine their behavior.

1 Introduction

There has recently been interest in investigating how the Unified Modeling Language (UML) [BJR98] can be applied to the modeling and analysis of agent-based software systems. For example, the OMG Agent Working Group [OMG00] is attempting to unify object-oriented approaches and current methodologies for developing agent systems. The result is a proposal for augmenting UML with agent-specific extensions (called Agent UML or AUML). This is based on the observation that the development of large-scale agent-based software requires modeling methods and tools that support the entire development life-cycle. Agent-based systems are highly concurrent and distributed and hence it makes sense to employ methodologies that have already been widely accepted for distributed object-oriented systems. Indeed, agents can be viewed as "objects with attitude" [Bra97] and can themselves be composed out of objects. On the other hand, agents have certain features not possessed by objects — such as autonomy, the ability to act without direct external intervention; and cooperation, the ability to independently coordinate with other agents to achieve a common purpose. The precise nature of the relationship between objects and agents is as yet unclear. However, we anticipate that the use of UML (perhaps with further extensions) for modeling agent-based systems will increase.

UML is essentially a collection of notations for modeling a system from different perspectives. Current commercial tools supporting UML (e.g., iLogix's Rhapsody [Rha99] and Rational's Rose [Rat99]) can generate C++ or Java code from statechart designs, but there is no support for translating between UML notations themselves. The focus of our own research is in bridging this gap, and to this end, we have developed an algorithm for translating UML sequence diagrams into UML statecharts. Sequence diagrams model message passing between objects, or in the case of agents, can be used to model communications between agents (i.e., agent interaction protocols). Statecharts take a state-centric view and model the behavior of a class of objects as a collection of concurrent, hierarchical finite state machines. In agent-based systems, our translation from sequence diagrams to statecharts can be used in:

– developing *agent skeletons* [Sin98a], which give abstract descriptions of agents in terms of the events that are significant for coordination with other agents. Agent skeletons are important in studying interactions in multi-agent systems. If conversation instances between agents are expressed as sequence diagrams, our translation algorithm can generate agent skeletons semi-automatically. A similar approach is followed in [Sin98b] except that conversations are described using Dooley graphs;
– developing behavioral models (i.e., statecharts) of agents that are composed of objects. In this context, our algorithm generates initial models from a collection of scenarios (sequence diagrams) of expected behavior. We view this process as being highly iterative — the generated statecharts will be refined by the user which feeds back to refined scenarios. Because of this iterative approach, the generated statecharts must be human readable — i.e., they must appropriately divide behavior into orthogonal components, and include sensible use of hierarchy.

Our techniques apply equally to agents and objects that make up agents. Hence, we will use 'agent' and 'object' interchangeably in what follows.

A number of other approaches have been developed for translating from scenarios to state machines (e.g., [KEK99,MST94,LMR98,SD95]), but our approach has a number of advantages, namely:

– Scenarios will in general overlap. Most other approaches cannot recognize intersections between scenarios. Our approach, however, performs a *justified merging of scenarios* based on logical descriptions of the communications between agents. The communications are specified using the Object Constraint Language (OCL) [WK99] and allow identical states in different scenarios to be recognized automatically. This leads to statecharts both with a vastly reduced number of states, and also corresponding more to what a human designer would produce.
– Scenarios will in general conflict with each other. Our algorithm first *detects and reports any conflicts* based on the specifications of the communications.
– The statecharts generated by our algorithm are *hierarchical and make sensible use of concurrency*. Much of this structure is detected automatically

from the communication specifications. Additional structure can be deduced from user-specified abstractions. This leads to generated statecharts that are human-readable, not just huge, flat state machines.

2 Example

We will use an ongoing example to illustrate our algorithm. The example is that of an automated loading dock in which forklift agents move colored boxes between a central ramp and colored shelves such that boxes are placed on shelves of the same color. The example is presented as a case study in [Mül96] of a three-layered architecture for agent-based systems, in which each agent consists of a reactive, a local planning and a coordination layer. Each layer has responsibility for certain actions: the reactive layer reacts to the environment and carries out plans sent from the planning layer; the planning layer forms plans for individual agent goals; and the coordination layer forms joint plans that require coordination between agents. We have translated part of this example into UML as a case study for our algorithm. Figure 1 gives the static structure of part of the system, represented as a UML class diagram. Each class can be annotated with attributes (typed in OCL) or associations with other classes. `coordWith` describes whether an agent is currently coordinating its actions with another agent (`0..1` is standard UML notation for multiplicity meaning 0 or 1), and `coordGoal` gives the current goal of this other agent. Agent interaction is based on a leader election protocol which selects an agent to delegate roles in the interaction. `leader` describes whether an agent is currently a leader. The filled diamonds in the class diagram represent aggregation (the 'part-of' relationship).

Figures 2, 3 and 4 are sample sequence diagrams (SDs) for interaction between two agents. SD1 is a failed coordination. Agent[i] attempts to establish a connection with Agent[j], but receives no response[1], so moves around Agent[j]. In SD2, the move is coordinated, and SD3 shows part of a protocol for Agent[j] to clear a space on a shelf for Agent[i]. Note that these are actually *extended* sequence diagrams. 'boxShelfToRamp' is a sub-sequence diagram previously defined and 'waiting' is a state explicitly given by the user. More will be said about extended SDs in Section 3.2.

3 Methodology

An increasingly popular methodology for developing object-oriented systems is that of use-case modeling [RS99], in which use-cases, or descriptions of the intended use of a system, are produced initially and are used as a basis for detailed design. Each use case represents a particular piece of functionality from a user perspective, and can be described by a collection of sequence diagrams. [RS99] advocates developing the static model of a system (i.e., class diagram) at the same time as developing the sequence diagrams. Once this requirements phase

[1] `tm` is a timeout

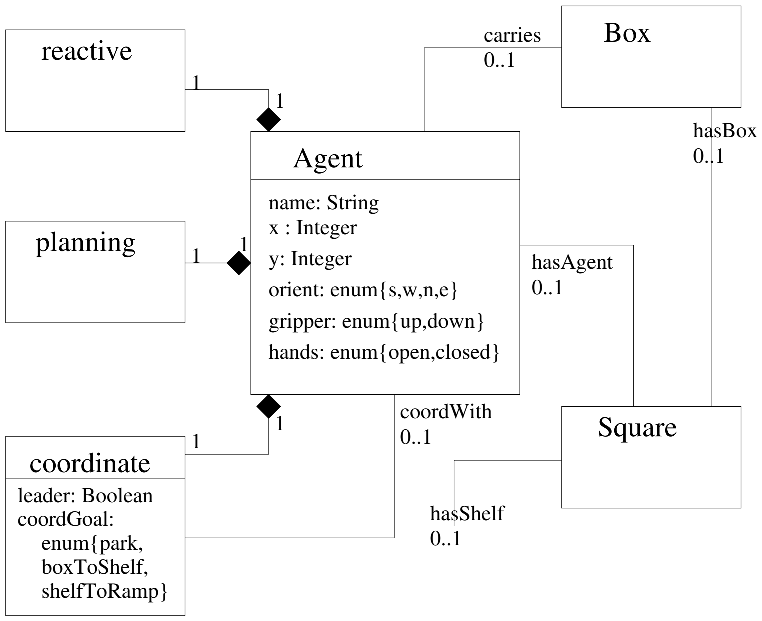


Fig. 1. The loading dock domain.

has been completed, more detailed design can be undertaken, e.g., by producing statecharts.

We leverage off this approach and our algorithm fits in as shown in Figure 5. From a collection of sequence diagrams, plus information from a class diagram and an OCL specification[2], a collection of statecharts is generated, one for each class. Note that the methodology is highly iterative — it is not expected that the designer gets the class diagram, sequence diagrams, or OCL specification correct first time. On the contrary, sequence diagrams will in general conflict with each other or the OCL spec, be ambiguous, or be missing information. The insertion of our algorithm enables some conflicts to be detected automatically and allows a much faster way of making modifications which will be reflected in the statechart designs. Note that one statechart is generated for each class.

3.1 OCL Specification

The lack of semantic content in sequence diagrams makes them ambiguous and difficult to interpret, either automatically or between different stakeholders. In current practice, ambiguities are often resolved by examining the informal documentation but, in some cases, ambiguities may go undetected leading to costly software errors. To alleviate this problem, we encourage the user to give pre/post-condition style OCL specifications of the messages passed between objects. These

[2] OCL is a side-effect free set-based constraint language.

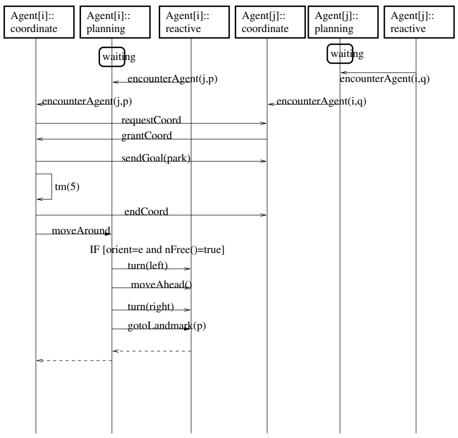


Fig. 2. Agent Interaction (SD1).

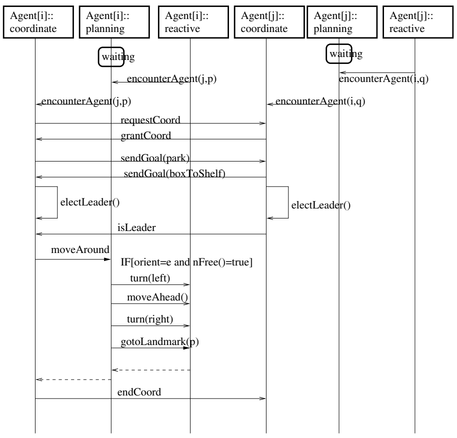


Fig. 3. Agent Interaction (SD2).

specifications include the declaration of *state variables*, where a state variable represents some important aspect of the system, e.g., whether or not an agent is coordinating with another agent. This OCL specification allows the detection of conflicts between different scenarios and allows scenarios to be merged in a *justified* way. Note that not every message needs to be given a specification, although, clearly, the more semantic information that is supplied, the better the quality of the conflict detection. Currently, our algorithm only exploits constraints of the form $var = value$, but there may be something to be gained from reasoning about other constraints using an automated theorem prover or model checker.

Figure 6 gives specifications for selected messages in our agents example. `Agent.coordWith` has type `Agent` (it is the agent which is coordinating with `Agent`), and `Agent.coordNum`, the number of agents `Agent` is coordinating with, is a new variable introduced as syntactic sugar.

The state variables, in the form of a *state vector*, are used to characterize states in the generated statechart. The state vector is a vector of values of the state variables. In our example, the state vector for the class `coordinate` has the form:

$$\langle\, \texttt{coordNum\^{}},\ \texttt{leader\^{}},\ \texttt{coordGoal\^{}} \,\rangle$$

where $var\hat{} \in Dom(var) \cup \{?\}$, and ? represents an unknown value. Note that since each class has a statechart, each class has its own state vector.

Our algorithm is designed to be fully automatic. The choice of the state vector, however, is a crucial design task that must be carried out by the user. The choice of state variables will affect the generated statechart, and the user should choose state variables to reflect the parts of the system functionality that is of most interest. In this way, the choice of the state vector can be seen as a powerful abstraction mechanism — indeed, the algorithm could be used in a way that allows the user to analyse the system from a number of different perspectives, each corresponding to a particular choice of state vector.

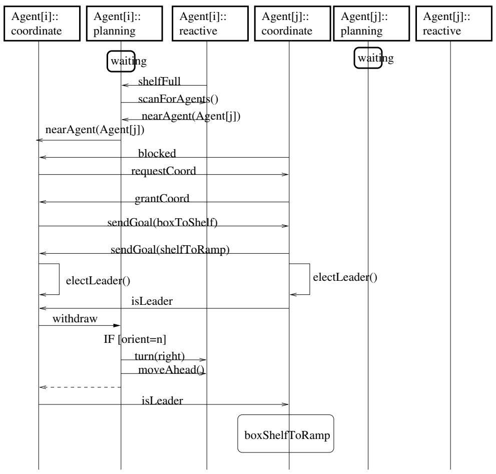


Fig. 4. Agent Interaction (SD3).

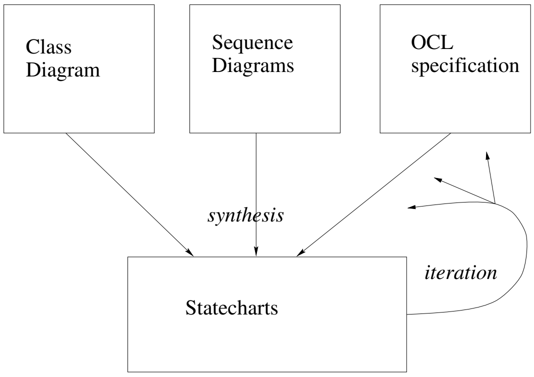


Fig. 5. Use-case modeling with Statechart Synthesis.

The state variables can be chosen from information present in the class diagram. For instance, in our example, the state variables are either attributes of a particular class or based on associations. The choice is still a user activity, however, as not all attributes/associations are relevant.

3.2 Extended Sequence Diagrams

Other authors ([GF99,BGH$^+$98]) have already noted that the utility of sequence diagrams to describe system behavior could be vastly increased by extending the notation. A basic SD supports the description of *exemplary* behavior — one concrete interaction — but when used in requirements engineering, a *generative* style is more appropriate, in which each SD represents a collection of interactions. Extensions that have been suggested include the ability to allow `case` statements,

```
coordNum : enum {0,1}
leader : Boolean
coordGoal : enum {park, boxToShelf, shelfToRamp}

context Agent.coordinate::grantCoord
   pre:  coordNum = 0 and coordWith.coordinate.coordNum = 0
   post: coordNum = 1 and coordWith.coordinate.coordNum = 1

context sendGoal(x : enum {park, boxToShelf, shelfToRamp})
   post: coordWith.coordinate.coordGoal = x

context electLeader
   pre:  leader = false

context isLeader
   post: coordWith.coordinate.leader = true

context endCoord
   pre:  coordNum = 1 and coordWith.coordinate.coordNum = 1
   post: coordNum = 0 and coordWith.coordinate.coordNum = 0
         and leader = false
```

Fig. 6. Domain Knowledge for the Loading Dock Example.

loops and sub-SDs. We go further than this and advocate the use of language constructs that allow behavior to be generalized. Example constructs we have devised so far include:

– $any\_order(m_1, \ldots, m_n)$: specify that a group of messages may occur in any order;
– $or(m_1, \ldots, m_n)$: a message may be any one of a group of messages;
– $generalize(m, SubSD)$: a message gives the same behavior when sent/ received at any point in the sub-sequence diagram;
– $allInstances(m, I)$: send a message to all instances in I;

These extensions significantly augment the expressiveness of sequence diagrams and their utility in describing system behaviors.

4 Generating Statecharts

Synthesis of statecharts is performed in four steps: first, each SD is annotated with state vectors and conflicts with respect to the OCL spec are detected. In the second step, each annotated SD is converted into flat statecharts, one for each class in the SD. The statecharts for each class, derived from different SDs, are then merged into a single statechart for each class. Finally, hierarchy is introduced in order to enhance readability of the synthesized statecharts.

4.1 Step I: Annotating Sequence Diagrams with State Vectors

The process to convert an individual SD into a statechart starts by detecting conflicts between the SD and the OCL spec (and hence, other SDs). There are two kinds of constraints imposed on a SD: constraints on the state vector given by the OCL specification, and constraints on the ordering of messages given by the SD itself. These constraints must be solved and conflicts be reported to the user. Conflicts mean that either the scenario does not follow the user's intended semantics or the OCL spec is incorrect.

More formally, the process of conflict detection can be written as follows. An annotated sequence diagram is a sequence of messages $m_1, \ldots, m_n$, with

$$s_0^{\mathsf{pre}} \xrightarrow{m_1} s_0^{\mathsf{post}}, s_1^{\mathsf{pre}} \xrightarrow{m_2} \ldots \xrightarrow{m_{r-1}} s_{r-1}^{\mathsf{post}}, s_r^{\mathsf{pre}} \xrightarrow{m_r} s_r^{\mathsf{post}} \tag{1}$$

where the s_i^{pre}, s_i^{post} are the state vectors immediately before and after message m_i is executed. S_i will be used to denote either s_i^{pre} or s_i^{post}; $s_i^{\mathsf{pre}}[j]$ denotes the element at position j in s_i^{pre} (similarly for s_i^{post}).

In the first step of the synthesis process, we assign values to the variables in the state vectors as shown in Figure 7. The variable instantiations of the initial state vectors are obtained directly from the message specifications (lines 1,2): if message m_i assigns a value y to a variable of the state vector in its pre- or post-condition, then this variable assignment is used. Otherwise, the variable in the state vector is set to an undetermined value ?. Since each message is specified independently, the initial state vectors will contain a lot of unknown values. Most (but not all) of these can be given a value in one of two ways: two state vectors, S_i and S_j ($i \neq j$), are considered the same if they are unifiable (line 6). This means that there exists a variable assignment ϕ such that $\phi(S_i) = \phi(S_j)$. This situation indicates a potential loop within a SD. The second means for assigning values to variables is the application of the frame axiom (lines 8,9), i.e., we can assign unknown variables of a pre-condition with the value from the preceeding post-condition, and vice versa. This assumes that there are no hidden side-effects between messages.

A conflict (line 11) is detected and reported if the state vector immediately following a message and the state vector immediately preceding the next message differ.

Example. Figure 8 shows SD2 from Figure 3 annotated with state vectors for `Agent`$[i]$`::coordinate`. Figure 9 shows how the state vectors are propagated.

4.2 Step II: Translation into a Finite State Machine

Once the variables in the state vectors have been instantiated as far as possible, a flat statechart (in fact, a finite state machine (FSM)) is generated for each class (or agent) in each SD. The finite state machine for agent A is denoted by Φ_A; its set of nodes by N_A; its transitions by $\langle n_1, \langle \mathit{type}, \mathit{label} \rangle, n_2 \rangle$ for nodes n_1,

Input. An annotated SD
Output. A SD with extended annotations

```
1  for each message $m_i$ do
2      if $m_i$ has a precondition $v_j = y$ then $s_i^{\mathsf{pre}}[j] := y$ else $s_i^{\mathsf{pre}}[j] := ?$ fi
3      if $m_i$ has a postcondition $v_j = y$ then $s_i^{\mathsf{post}}[j] := y$ else $s_i^{\mathsf{post}}[j] := ?$ fi
4  for each state vector $S$ do
5      if there is some $S' \neq S$ and some unifier $\phi$ with $\phi(S) = \phi(S')$ then
6          unify $S_i$ and $S_j$;
7          propagate instantiations with frame axiom:
8          for each $j$ and $i > 0$ : if $s_i^{\mathsf{pre}}[j] = ?$ then $s_i^{\mathsf{pre}}[j] := s_{i-1}^{\mathsf{post}}[j]$ fi
9                                     if $s_i^{\mathsf{post}}[j] = ?$ then $s_i^{\mathsf{post}}[j] := s_i^{\mathsf{pre}}[j]$ fi
10     if there is some $k, l$ with $s_k^{\mathsf{post}}[l] \neq s_{k+1}^{\mathsf{pre}}[l]$ then
11         Report Conflict;
12         break;
```

Fig. 7. Extending the state vector annotations.

n_2 where *type* is either *event* or *action*[3]; and μ_A is a function mapping a node to its state vector. $\mathcal{C}_\mathcal{A}$ denotes the currently processed node during the run of the algorithm. Messages directed towards a particular agent, A (i.e., $m_i^{to} = A$) are considered events in the FSM for A. Messages directed away from A (i.e., $m_i^{from} = A$) are considered actions.

The algorithm for this synthesis is depicted in Figure 10. Given a SD, the algorithm constructs one FSM for each agent (or for each class, in case we consider agents consisting of objects) mentioned in the sequence diagram. We start by generating a single starting node $n_{A_i}^0$ for each FSM (line 2). Then we successively add outgoing and incoming messages to the FSMs, creating new nodes as we proceed (lines 7-9).

An important step during FSM creation is the identification of loops: a loop is detected if the state vector immediately after the current message has been executed is the same as an existing state vector *and* if this message is state-changing, i.e., $s_i^{\mathsf{pre}} \neq s_i^{\mathsf{post}}$. Note that some messages may not have a specification, hence they will not affect the state vector. To identify states based solely on the state vector would result in incorrect loop detection.

4.3 Step III: Merging Multiple Sequence Diagrams

The previous steps concerned the translation of a single SD into a number of statecharts, one for each class. Once this is done for each SD, there exists a

[3] In statecharts, a transition is labeled by e/a which means that this transition can be active only if event e occurs. Then, the state changes and action a is carried out. We use a similar notion in our definition of FSMs.

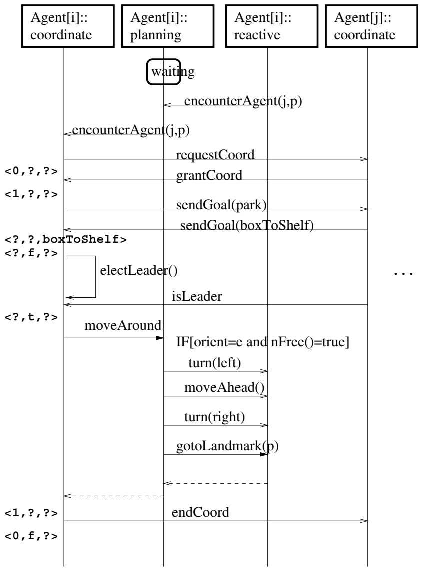


Fig. 8. SD2 (parts) with state vectors ⟨ `coordNum^`, `leader^`, `coordGoal^` ⟩.

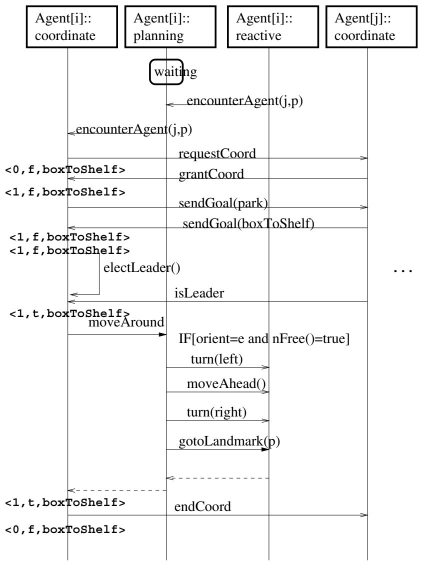


Fig. 9. SD2 after extension of state vector annotations.

collection of flat statecharts for each class. We now show how the statecharts for a particular class can be merged.

Merging statecharts derived from different SDs is based upon identifying *similar* states in the statecharts. Two nodes of a statechart are *similar* if they have the same state vector and they have at least one incoming transition with the same label. The first condition alone would produce an excessive number of similar nodes since some messages do not change the state vector. The additional required existence of a common incoming transition means that in both cases, an event has occurred which leaves the state variables in an identical assignment. Hence, our definition of similarity takes into account the ordering of the messages and the current state. Figure 11 shows how two nodes with identical state vector S and incoming transitions labeled with l can be merged together.

The process of merging multiple statecharts proceeds as follows: we generate a new statechart and connect its initial node by empty ϵ-transitions with the initial nodes of the individual statecharts derived from each SD. Furthermore, all pairs of nodes which are *similar* to each other are connected by ϵ-transitions. Then we remove ϵ-transitions, and resolve many non-deterministic branches. For this purpose, we use an algorithm which is a variant of the standard algorithm for transforming a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA) [ASU86][4].

[4] Note that the output of the algorithm is only deterministic in that there are no ϵ-transitions remaining. There may still be two transitions leaving a state labelled

```
Input. A SD, $S$, with agents $A_1, \ldots, A_k$ and messages $m_1, \ldots, m_r$
Output. A FSM $\Phi_{A_i}$ for each agent, $1 \leq i \leq k$.

for $i = 1, \ldots, k$ do
    Create a FSM, $\Phi_{A_i}$, with a initial node, $n^0_{A_i}$; $\mathcal{C}_{A_i} := n^0_{A_i}$;
for $i = 1, \ldots, r$ do
    ADD($m_i, action, m_i^{from}$);
    ADD($m_i, event, m_i^{to}$);
where ADD(mess $m$, type $t$, agent $A$)
    if there is a node $n \in N_A$, a transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$
      and $s_i^{\mathsf{post}} = \mu_A(n)$ then
        $\mathcal{C}_A := n$;
    else if there is $n \in N_A$ with $s_i^{\mathsf{post}} = \mu_A(n)$
      and $m_i$ is state-changing then
        add new transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$;
        $\mathcal{C}_A := n$;
    else
        add a new node $n$ and let $\mu_A(n) := s_i^{\mathsf{post}}$;
        add transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$;
        $\mathcal{C}_A := n$;
```

Fig. 10. Translating a sequence diagram into FSMs.

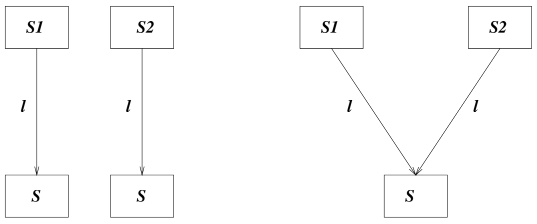


Fig. 11. Merging of similar states (before and after the merge). .

5 Introducing Hierarchy

So far, we have shown how to generate FSMs without any hierarchy. In practice, however, statechart designs tend to get very large and so the judicious use of hierarchy and orthogonality is crucial to the readability and maintainability of the designs. Our approach allows several ways for introducing hierarchy into the generated FSMs: using implicit information present in the state vectors, introducing generalizations, and using information explicitly given by the user (e.g., in a UML class diagram). These techniques will be discussed in the following.

with the same events but different actions. Hence, our algorithm may produce non-deterministic statecharts, which allows a designer to refine the design later.

5.1 Using the State Vector

State variables usually encode that the system is in a specific mode or state (e.g., holding a box or not). Thus it is natural to partition the statechart into subcharts containing all nodes belonging to a specific mode of the system. More specifically, we recursively partition the set of nodes according to the different values of the variables in the state vectors. In general, however, there are many different ways of partitioning a statechart, not all of them suited for good readability.

Thus, we introduce additional heuristic constraints (controlled by the user) on the layout of the statechart:

1. the maximum depth of hierarchy. Too many nested levels of compound states limit readability of the generated statechart. On the other hand, a statechart which is too flat contains very large compound nodes, making reading and maintaining the statechart virtually impossible.
2. the maximum number of states on a single level. This constraint is orthogonal to the first one and also aims at generating "handy" statecharts.
3. the maximum percentage of inter-level transitions. Transitions between different levels of the hierarchy usually limit modularity, but occasionally they can be useful. Thus their relative number should be limited (usually around 5-10%).
4. a partial ordering over the state variables. This ordering describes the sequence in which partitions should be attempted. It provides a means to indicate that some state variables may be more "important" than others and thus should be given priority. The ordering encapsulates important design decisions about how the statechart should be split up.

In general, not all of the above constraints can be fulfilled at the same time. Therefore our algorithm has to have the capability to do a search for an optimal solution. This search is done using backtracking (currently over different partial orders) and is strictly limited to avoid excessive run-times.

Example. Figure 12 gives a partitioned statechart for agent communication generated from SD1, SD2 and SD3. The flat statechart was first split over `coordNum`, followed by `leader` and finally `coordGoal`.

5.2 Hierarchy by Language Constructs

Section 3.2 introduced extended sequence diagrams. The actual constructs used in these sequence diagrams can also be used to introduce hierarchy into the generated statechart. As an example, $any\_order(m_1, \ldots, m_n)$ can be implemented by n concurrent statecharts (see Figure 13), connected by the UML synchronization operator (the black bar) which waits until all its source states are entered before its transition is taken. This is particularly useful if $m_1, \ldots, m_n$ are not individual messages, but sub-sequence diagrams. Figure 13 shows how the other constructs mentioned in Section 3.2 can be implemented as statecharts. *allInstances* is implemented by a local variable that iterates through each instance, and sends the message to that instance.

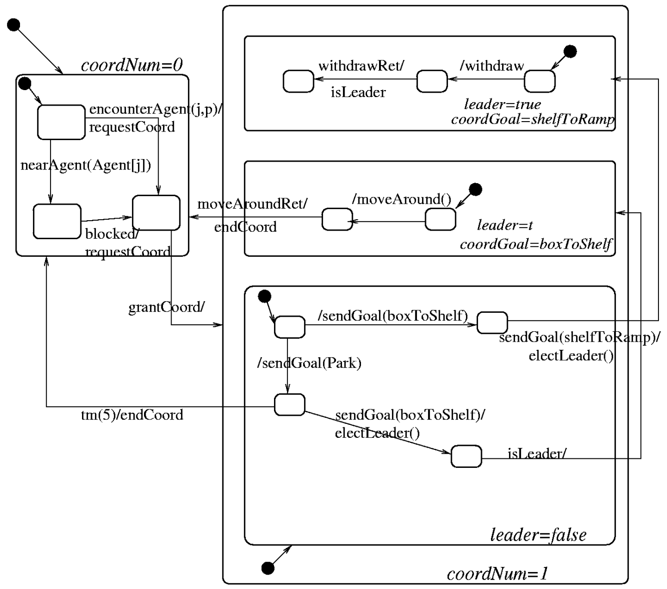


Fig. 12. Hierarchical Statechart for Agent::coordinate.

6 Conclusions

In this paper, we have presented an algorithm for automatically generating UML statecharts from a set of sequence diagrams. For the development of large-scale agent-based systems, sequence diagrams can be a valuable means to describe inter-agent communication. We extend sequence diagrams with additional language constructs to enable generalizations and augment them with communication pre- and post-conditions in OCL. This enables us to automatically detect and report conflicts between different sequence diagrams and inconsistencies with respect to the domain theory. These annotations are furthermore used in our algorithm to correctly identify similar states and to merge a number of sequence diagrams into a single statechart. In order to make the algorithm practical, we introduce hierarchy into the statechart.

A prototype of this algorithm has been implemented in Java and so far used for several smaller case-studies in the area of agents, classical object-oriented design [WS00], and human-computer interaction. In order to be practical for applications on a larger scale, our future work includes a tight integration of our algorithm into a state-of-the-art UML-based design tool.

The algorithm described in this paper only describes the forward or synthesis part of the design cycle: given a set of SDs, we generate a statechart. For full support of our methodology, research and development in two directions are of major importance: conflicts detected by our algorithm must not only be reported in an appropriate way to the designer but also should provide explana-

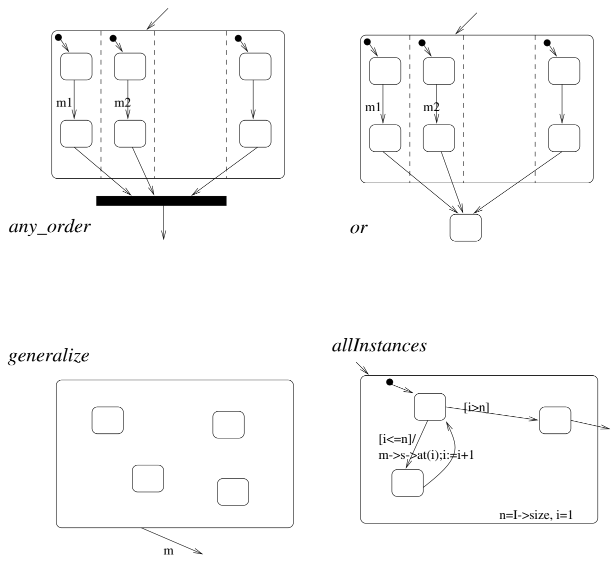


Fig. 13. Hierarchy by Macro Commands.

tion on what went wrong and what could be done to avoid this conflict. We will use techniques of model-generation, abduction, and deduction-based explanation generation to provide this kind of feedback.

The other major strand for providing feedback is required when the user, after synthesizing the statechart, refines it or makes changes to the statechart. In that case, it must be checked if the current statechart still reflects the requirements (i.e., the sequence diagrams), and in case it does, must update the sequence diagrams (e.g., by adding new message arrows).

The question whether UML (or AUML) is an appropriate methodology for the design of large-scale agent-based systems must still be answered. A part of the answer lies in the availability of powerful tools which facilitate the development of agents during all phases of the iterative life-cycle. We are confident that our approach to close the gap between requirements modeling using sequence diagrams and design with statecharts will increase acceptability of UML methods and tools for the design of agent-based systems.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

[BGH+98] R. Breu, R. Grosu, C. Hofmann, F. Huber, I. Krüger, B. Rumpe, M. Schmidt, and W. Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Computer Standards and Interfaces*, volume 19, pages 335–345, 1998.

[BJR98] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.

[Bra97] J. Bradshaw. *Software Agents*. American Association for Artificial Intelligence / MIT Press, 1997.

[GF99] T. Gehrke and T. Firley. Generative Sequence Diagrams with Textual Annotations. In Spies and Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme (FBT99) (Formal Description Techniques for Distributed Systems)*, pages 65–72, München, 1999.

[KEK99] I. Khriss, M. Elkoutbi, and R.K. Keller. Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams. In J. Bezivin and P.A. Muller, editors, *UML98: Beyond the Notation*, pages 132–147. Springer-Verlag, 1999. LNCS 1618.

[LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing Software Architecture Descriptions from Message Sequence Chart Specifications. In *Automated Software Engineering*, pages 192–195, Honolulu, Hawaii, 1998.

[Mül96] J. Müller. *The Design of Intelligent Agents*. Springer, 1996. LNAI 1177.

[MST94] T. Männistö, T. Systä, and J. Tuomi. SCED Report and User Manual. Report A-1994-5, Dept of Computer Science, University of Tampere, 1994. ATM example available with the SCED tool from `http://www.uta.fi/~cstasy/scedpage.html`.

[OMG00] OMG Agent Working Group. Agent Technology Green Paper. Technical Report ec/2000-03-01, Object Management Group, March 2000.

[Rat99] *Rational Rose*. Rational Software Corporation, Cupertino, CA, 1999.

[Rha99] *Rhapsody*. I-Logix Inc., Andover, MA, 1999.

[RS99] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML*. Object Technology Series. Addison Wesley, 1999.

[SD95] S. Somé and R. Dssouli. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *Asia Pacific Software Engineering Conference*, pages 48–57, 1995.

[Sin98a] M. Singh. A Customizable Coordination Service for Autonomous Agents. In *Intelligent Agents IV: 4th International Workshop on Agent Theories, Architectures, and Languages*, 1998.

[Sin98b] M. Singh. Developing Formal Specifications to Coordinate Heterogeneous Autonomous Agents. In *International Conference on Multi Agent Systems*, pages 261–268, 1998.

[WK99] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[WS00] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000.

Direct Execution of Agent Specifications

Michael Fisher

Centre for Agent Research and Development, Department of Computing and Mathematics
Manchester Metropolitan University, United Kingdom [http://www.card.mmu.ac.uk]

Although agent-based systems are beginning to be used in critical applications, it is clear that more precise, and logically well-founded, development techniques will be required for agent-based applications in the future. Thus, our aim is to provide a formal framework supporting the principled development of agent-based systems, and comprising: logics in which the high-level behaviours (of both agent and system) can be concisely specified; techniques for refinement and verification of such specifications; and a programming language providing concepts close to the specification notation used. While the specification and verification aspects are covered elsewhere, this talk concerns the programming of agent-based systems based upon the direct execution of specifications. The core formalism we use is *temporal logic*, which is both simple and intuitive, yet powerful. This enables us to specify the basic dynamic attributes of both the agent and the multi-agent environment. Thus, the basic specification of an agent's behaviour is given as a temporal formula, then transformed into a simple normal form, called Separated Normal Form (SNF), which provides a simple and intuitive description of what is true at the beginning of execution, what must be true during any execution step, and what constraints exist on future execution states [3]. Given an agent specified using SNF, we use a *forward-chaining* process on a set of SNF clauses in order to construct a model for the specification, and thus animate the agent specification [2]. Further, temporal specifications of the above form can be extended with aspects of deliberation [4] and with a dimension of (bounded) belief [6]. In addition to specifying and implementing individual agents, it is clearly important to be able to handle multi-agent systems. Thus, the above approach is being extended to a multi-agent environment in which asynchronously executing agents broadcast messages to each other [1]. A key aspect of this is the representation and implementation of both organisational structures [7] and environmental conditions within those structures [5].

References

1. M. Fisher. A Survey of Concurrent MetateM — The Language and its Applications. In *Proc. International Conference on Temporal Logic*, Springer-Verlag (LNCS 827), 1994.
2. M. Fisher. Representing and Executing Agent-Based Systems. In *Intelligent Agents*. Springer-Verlag (LNAI 890), 1995.
3. M. Fisher. A Normal Form for Temporal Logic and its Application in Theorem-Proving and Execution. *Journal of Logic and Computation*, 7(4), August 1997.
4. M. Fisher. Implementing BDI-like Systems by Direct Execution. In *Proc. International Joint Conference on Artificial Intelligence*. Morgan-Kaufmann, 1997.
5. M. Fisher. Representing Abstract Agent Architectures. In *Intelligent Agents V*. Springer-Verlag (LNAI 1555), 1999.
6. M. Fisher and C. Ghidini. Programming Resource-Bounded Deliberative Agents. In *Proc. International Joint Conference on Artificial Intelligence*. Morgan-Kaufmann, 1999.
7. M. Fisher and T. Kakoudakis. Flexible Agent Grouping in Executable Temporal Logic. In *Proc. International Symposium on Languages for Intensional Programming*, 1999.

Using the π-Calculus to Model Multiagent Systems [⋆]

Albert C. Esterline and Toinette Rorie[⋆⋆]

Department of Computer Science/NASA ACE
North Carolina A&T State University
Greensboro, NC 27411
esterlin@ncat.edu and rorie@lucent.com

Abstract. We present a formal framework that uses the π-calculus for modeling multiagent systems. A process algebra in general is a term algebra used as an abstract programming language that stresses the composition of processes by a small set of process operators. The π-calculus in particular allows one to express systems of processes that have changing communication structure. We explicate the agent abstraction as a π-calculus process that persists through communication actions. Our principal task here is to show how the π-calculus can be used to model certain aspects that have already been specified for a major multiagent system. We also sketch how a π-calculus framework supports development activities in this context, and we suggest how various general aspects of multiagent systems may be modeled in this framework.

1 Introduction

Wooldridge [20] defines an agent as a computer system capable of autonomous action. An *intelligent* agent in particular is capable of flexible autonomous action, where flexibility involves reactivity (the ability to perceive its environment and to respond to changes in it), pro-activeness (aiming at goals by taking the initiative), and social ability (since agents must cooperate to achieve goals). Autonomy distinguishes agents from objects since the decision whether to execute an action lies with the object invoking the method in an abject system but with the agent receiving the request in an agent system. This paper focuses on multiagent systems, which "provide an infrastructure specifying communication and interaction protocols" and are typically open (with no "centralized designer") [7]. Abstractly, the open nature of these environments entails that smaller systems (agents) can be *composed* to form larger systems (multiagent systems), where composition involves coordinated concurrent activity. Coordination is also critical and, in the case of self-interested agents, a major challenge.

⋆ This research was supported by grant NAG 5-4102, "Formal Foundations of Agents," from NASA Goddard Space Flight Center. Special thanks are due to the Goddard personnel, especially Walt Truszkowski.

⋆⋆ Current address: Lucent Technologies, 200 Lucent Lane, Cary, NC 27511

In modeling multiagent systems, we use the *agent abstraction*: the notion of an autonomous, flexible computational entity in an open environment that specifies communication protocols. This notion provides higher-level abstractions for complex systems, leading to simpler development techniques that alleviate the complexity of multiagent systems [17]. A *model* is (partially) expressed by a specification when our goal is system development. A (modeling) *framework* is a notation along with directions on using it to express (and hence to construct) models. This paper suggests some aspects of a formal framework for modeling such systems (without actually developing the framework) and, in particular, formally explicates the agent abstraction. Formal modeling naturally lags quick-and-dirty approaches to system design, but once a field has been defined in practice, formal modeling offers rigor, eliminates ambiguity, and allows one to abstract away inessential details.

Among concurrency formalisms, the most abstract are (temporal and modal) logics, used to describe or specify the communication behavior of processes. More concrete are process algebras, which are term algebras used as abstract concurrent programming languages that stress the composition of processes by a small set of process operators. The most concrete are concurrency automata (paradigmatically Petri nets, but also, for example, Statecharts), which describe processes as concurrent and interacting machines with all the details of their operational behavior. We have used modal logics to formalize the transaction abstraction for multiagent systems [9,18], high-level Petri nets to model agent behavior [2], and Statecharts to express multiagent plans [21]. These formalisms do not directly represent agents as complex entities, and they are not compositional. The possible exception is Statecharts, which have been given compositional, process-algebraic semantics; Statecharts have the advantages and disadvantages of a visual formalism, and there is some promise in using them to supplement process algebras.

There are several reasons to consider an agent as a process (as understood in concurrency theory, i.e., an entity designed for a possibly continuous interaction with its users) and to model multiagent systems with a process-algebraic framework. Compositionality allows process terms for agents to be combined simply to form a term for a multiagent system. Communication protocols required of a multiagent environment can be formulated straightforwardly in process algebras, which are often used for formalizing protocols. The basic action in a process algebra is communication across an interface with a *handshake*. Two processes performing a handshake must be prepared at the time to perform complementary actions (usually thought of as output and input on a given channel), hence a handshake synchronizes the processes. Handshakes are remarkably like speech acts in face-to-face communication. The symmetry of a handshake distributes control between the participating processes hence respects their autonomy. Since their are handshakes with the environment, the reactive nature of agents can be accommodated. A process-algebraic framework does not so directly capture aspects of the agent abstraction that are not directly related to communication. The pro-active aspect involves (internal) planning, and negotiation involves (internal) computation by the negotiators in addition to their communication. Process algebras, however, typically allow "silent" actions (handshakes not observable outside the process making them) among components that have had their communication ports in some sense internalized within a single process. Sequences of silent actions offer hope for modeling computations internal to agents.

But not all processes are agents. Generally, a process P can perform a communication action, or *transition*, then behave like the process resulting from reducing P in a certain way. Some of the operators (and the syntactic patterns they govern) persist through transitions; an example is the (parallel) composition operator. Others, such as the alternative operator, do not thus persist—once one alternative is selected, the others are no longer available. Think of a system as an encompassing process whose several component processes– agents—persist (perhaps in reduced form) through transitions. We picture a pair of agents P and Q as connected by a link—a possibility for a handshake—when P contains a name denoting an action and Q contains a name denoting the complementary action. This framework, then, gives a picture of the communication linkage of a multiagent system.

In interesting multiagent systems, however, this linkage generally changes dynamically—consider delegation of tasks or subtasks and the changing relations a mobile agent has to stationary agents as it moves from one region to another. Thus, the process albegra we use in our framework is the π-calculus, which allows *mobility*, the communication of a link as data in a handshake. The π-calculus has the advantage over other process algebras supporting mobility that it allows us to simulate higher-order features (passing processes as data) by simply passing links. In particular, link passing can model a system whose agent population changes since an agent that is not yet or no longer part of the system is one that has no links to agents currently in the system. Also, it has been shown that the π-calculus is a universal model for computation, so we can *in principle* model with it any computational aspect of agents.

The principal task of this paper is to show how the π-calculus can be used to model certain aspects that have already been specified for a major multiagent system, the LOGOS agent community being designed at NASA Goddard Space Flight Center. We capture aspects sufficient to support a scenario that has been supplied by NASA/GSFC; we are concerned with the branch of the scenario that requires dynamic change in the communication structure. Section 3 of this paper outlines the LOGOS agent community and presents the scenario, and section 4 gives a π-calculus specification for supporting the scenario; the π-calculus is presented in section 2. Section 5 suggests how a π-calculus framework supports development activities in this context. Building on the experience gained in this exercise, section 6 briefly addresses aspects of multiagent systems from the point of view of our framework. Section 7 concludes and suggests future work.

2 The π-Calculus

The π-calculus is a process algebra evolved from CCS [11] and notable in supporting mobility. There are two versions of the π-calculus: the monadic calculus [14], where exactly one name is communicated at each handshake, and the polyadic calculus [12], where zero or more names are communicated. This presentation mainly follows the monadic π-calculus because it is closer to CCS, with which many are already familiar and for which there are mature tools. Later (in sections 5 and 6), however, we shall cite results derived chiefly in the polyadic calculus, and Milner's new text [13] promises to make the polyadic calculus better known.

The basic concept behind the π-calculus is naming or reference. Names are the primary entities, and they may refer to links (or any other kind of basic entity). Processes, sometimes referred to as agents, are the only other kind of entities. (We tend to avoid the term "agent" in describing the π-calculus so as not to beg questions.) We let lower case letters from the end of the alphabet (e.g., x, y) range over the names and upper case letters from the beginning of the alphabet (e.g., A) range over process identifiers. Also, upper case letters from the middle of the alphabet (e.g., P, Q) range over process expressions, of which there are six kinds (corresponding to the six kinds of combinators or operators).

1. A *summation* has the form $\sum_{i \in I} P_i$, where I is a finite index set. This behaves like one or another of the P_i. The empty summation, or inaction, represented by $\mathbf{0}$, is the agent that can do nothing. The binary summation is written as $P_1 + P_2$.
2. A *prefix* is of the form $\overline{y}x.P$, $y(x).P$, or $\tau.P$. "$\overline{y}x.P$" is a *negative prefix*; $\overline{y}$ can be thought of as an output port of a process that contains it. $\overline{y}x.P$ outputs x on port $\overline{y}$ then behaves like P. "$y(x).P$" is a *positive prefix*, where y is the input port of a process; it binds the variable x. At port y the arbitrary name z is input by $y(x).P$, which behaves like $P\{z/x\}$, where $P\{z/x\}$ is the result of substituting z for all free (unbound) occurrences of x in P. We think of the two complementary ports y and $\overline{y}$ as connected by a channel (link), also called y. τ is the *silent action*; $\tau.P$ first performs the silent action and then acts like P.
3. A (parallel) *composition*, $P_1 \mid P2$, is a process consisting of P_1 and P_2 acting in parallel.
4. A *restriction* $(x)P$ acts like P and prohibits actions at ports x and $\overline{x}$, with the exception of communication between components of P along the link x. Restriction, like positive prefix, binds variables.
5. $[x = y]P$ is a *match*, where a process behaves like P if the names x and y are identical.
6. A *defined agent* (with arity n), $A(y_1 \ldots, y_n)$, has a unique defining equation, $A(x_1 \ldots, x_n) =_{def} P$, where $x_1 \ldots, x_n$ are distinct names, the only ones that may occur free in P. $A(y_1 \ldots, y_n)$ behaves like $P\{y_1/x_1 \ldots, y_n/x_n\}$ for the simultaneous substitution of y_i for all free occurrences of $x_i (1 \leq i \leq n)$ in P.

A process P may perform a communication action (transition) α, corresponding to a prefix, and evolve into another process P'. This is indicated by the notation $P \overset{\alpha}{\to} P'$. The meanings of the combinators are formally defined by transition rules, each with a conclusion (stated below a line) and premises (above the line). The following rule is of particular interest.

$$\frac{P \overset{\overline{x}y}{\to} P',\ Q \overset{x(z)}{\to} Q'}{P \mid Q \overset{\tau}{\to} P' \mid Q'\{y/z\}}$$

This indicates communication, or handshake, between P and Q, resulting in a silent action, τ, whereby a name, y, is communicated between P (now P') and Q (now Q'). For example,

$$\overline{y}x.P \mid y(z).Q \ \overset{\tau}{\to}\ P \mid Q\{x/z\}$$

so that after this communication all free occurrences of z in Q are replaced by whatever value x had in $\overline{y}x.P$. Of particular interest are those cases where we use restriction to internalize ports so that they cannot be observed from the outside, forcing communication actions to match their complements. For example, $(y)\,(\overline{y}x.P \mid y(z).Q)$ can perform only a τ action where x is communicated along channel y to become the value of z in Q.

Consider a simple case of restructuring a system of processes. We begin with

$$(y)(m)\,(y(x).\overline{x}z.\mathbf{0} \mid \overline{y}m.\mathbf{0} \mid m(u).P)$$

with three parallel components. Because of the restrictions on channels y and m, the only action that can occur in this system is a τ action by which the second component sends m on channel y to the first component. We then have $(m)(\overline{m}\,z.\mathbf{0} \mid m(u).P)$. The second component has disappeared, being reduced to the do-nothing process $\mathbf{0}$. Also, the restriction (y) has been dropped since y no longer occurs in the expression. The second component communicated to the first a channel, m, and the only action the resulting system can perform is a τ action by which z is communicated along this channel, resulting in $P\{z/u\}$.

3 The LOGOS Multiagent System

The Lights Out Ground Operations System (LOGOS) [10] is a multiagent system in its initial stages of development at the Automation Technology Section of the NASA Goddard Space Flight Center (NASA/GSFC). It is a prototype for an unattended grounds operation center whose main objectives are to demonstrate the integration of existing technologies, to deploy and evaluate evolving agent concepts in the context of lights-out operations, to support the evaluation of data visualization ideas and cognitive studies, and to establish an environment that supports the development of multiagent systems. A LOGOS agent acts as a "substitute controller" to link together ground center technologies. These technologies include GenSAA and Genie (Generic Spacecraft Analyst Assistant and Generic Inferential Executor), which graphically build expert systems that have the ability to monitor command spacecraft operations. The agents communicate via asynchronous message passing, and an agent communication language is used to define the message structure and content.

We only model the communication that takes place between agents that make up a subpart of the LOGOS system. This *anomaly system* involves five agents (FIRE, GIFA, DBIFA, UIFA, and PAGER) responsible for resolving detected mission faults within a simulated spacecraft. These agents effectively resolve faults by coordinating analysis activities and tracking the progress of fault resolution activities. We focus primarily on FIRE since it is responsible for resolving the detected mission faults. FIRE (Fault Isolation and Resolution Expert) [15] is responsible for providing procedures for resolving mission faults reported by GenSAA or requesting the direct intervention of a human expert to isolate and resolve them. FIRE is also responsible for supporting the human expert. The information passed between FIRE and the other four agents is shown in the contextual diagram in Figure 1. DBIFA is responsible for storing and retrieving

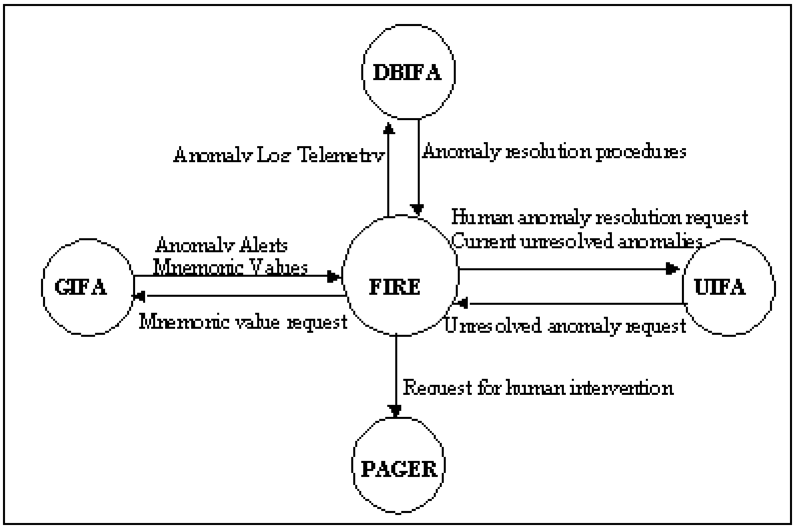

DBIFA
Anomaly Log Telemetry
Anomaly resolution procedures
Human anomaly resolution request
Current unresolved anomalies
Anomaly Alerts
Mnemonic Values
GIFA
FIRE
UIFA
Mnemonic value request
Unresolved anomaly request
Request for human intervention
PAGER

Fig. 1. FIRE Contextual Diagram

information into and from databases, such as mnemonic metadata, anomaly information, and anomaly resolution procedures. Upon receipt of FIRE's request for matching fault resolution procedures, DBIFA searches for these procedures. If it succeeds, it sends FIRE the procedures. If, however, the search is unsuccessful, FIRE invokes a more elaborate isolation process. GIFA (GenSAA Interface Agent) is a reactive agent responsible for broadcasting anomaly alerts and providing real-time telemetry. It monitors the GenSAA Data Server for any anomalies, alerts FIRE to any anomaly occurrences, and provides FIRE with mnemonic values or inferred variable values upon request. UIFA (User Interface Agent) responds to an agent request or any other type of data that it receives from the user interface, informing the user of anomalies and schedule changes. When UIFA receives an anomaly alert and a user is not logged on, a message is sent to PAGER to page the appropriate system personnel. We focus on the branch of a scenario supplied by NASA/GSFC of the anomaly system. This branch, where FIRE is unable to resolve the fault and requests UIFA for human assistance, involves mobility. Each step is labeled since it will be identified with one or more π-calculus process definitions.

A. GIFA to FIRE Alerts FIRE that a fault has occurred.
B. FIRE to GIFA Requests GIFA for the mnemonic values associated with the fault.
C. GIFA to FIRE Informs FIRE of the mnemonic values associated with the fault.
D. FIRE to DBIFA Requests DBIFA for fault resolution procedures for resolving the fault.
E. DBIFA to FIRE Sends FIRE matching fault resolution procedures or informs it that no procedures were found.

In this branch of the scenario, FIRE is unable to resolve the fault and requests UIFA for human assistance.

F. FIRE to UIFA Requests UIFA for human intervention in resolving the fault.
G. UIFA to FIRE Responds to FIRE that a user will resolve the fault or that no user is present.
H. FIRE to PAGER Requests PAGER to locate a user.
I. PAGER to UIFA Signals UIFA to resolve a fault.
J. UIFA to PAGER Commands PAGER to stop paging.
K. UIFA to FIRE Responds to FIRE that it will resolve the fault.

In the next section, we present a π-calculus specification of the anomaly system that captures the behavior presented here. The specification makes explicit much of the understanding one has of the system before analyzing the scenario.

4 The π-Calculus Specification for the Scenario

In modeling a system that may support the above scenario, we first identify the communication links required. Table 1 gives, for each link, its mnemonic name, the agent ("Sender") at the output port of the link, the agent ("Receiver") at the input port, and the kind of message passed by a handshake on the link. Next, we develop a *flowgraph* of the system being modeled. A flowgraph depicts the linkage among a system's components, not its dynamic properties. A node in a flowgraph is a black box with "buttons," that is, ports that appear in the process expressions. A complementary pair $(x, \overline{x})$ of ports represents a means of

Table 1. The Links for the Anomaly System

Name	Sender	Receiver	Message Type
mvreq	FIRE	GIFA	mnemonic value request
amsg	GIFA	FIRE	advisory message
areq	FIRE	DBIFA	request for resolution procedures
matproc	DBIFA	FIRE	matching resolution procedures
humreq	FIRE	UIFA	request for a human to resolve the fault
humres	UIFA	FIRE	response from the human
reqhum	FIRE	PAGER	request for a human to resolve the fault
sighum	PAGER	UIFA	signal to contact the human
respsig	UIFA	PAGER	response to request for human intervention

interaction between black boxes. Such a pair of ports is connected in the flowgraph by an edge representing the corresponding link or channel. In a flowgraph that corresponds to a multiagent system, it is natural to have exactly one node for each agent. We get the flowgraph for the anomaly system by taking the contextual diagram in Figure 1 and using the link names in Table 1. Since the links *sighum* and *respsig* are passed to UIFA and PAGER, this flowgraph changes in the course of the scenario to include these links and their corresponding ports. We would not generally know in advance that there would be a linkage between $UIFA$ and $PAGER$ since there are generally several ways in which the linkage of a system may evolve. Here, however, we are modeling a system only to the extent that it supports one branch in a scenario (where linkage is established). Furthermore, we have simplified the situation by assuming that only the link *sighum* is available for PAGER-to-UIFA communication and only the link *respsig* is available for UIFA-to-PAGER communication. A more complex setting would admit several pager agents and several user interface agents and may also admit several links in both directions. Summation captures straightforwardly any indifference in the model to choices among the resources.

Since the anomaly system encompasses the five processes shown in Figure 1 operating in parallel, the process expression for the entire system should include the composition

$$FIRE \mid GIFA \mid DBIFA \mid UIFA \mid PAGER$$

All we need add to this is restrictions that bind the nine links (see Table 1) that will appear in the process expressions that elaborate the five processes shown in the flowgraph. The process expression for the entire community would be a composition like the above but containing considerably more process identifiers. The scope of a given restriction in this expression would not necessarily include the entire composition since we may not want certain ports to be visible to certain agents, and we may want to conceptualize the system as partitioned into subsystems. A link local to a subsystem would be bound by a restriction whose scope is the composition representing the subsystem.

Here we define most π-calculus processes without parameters since the names being substituted for are generally bound by positive prefixes (they receive input values). A typical, if simple, example of a process definition of this form is

$$A =_{def} x(y).A1$$

Here, after y is bound to a value, A behaves like $A1$, which apparently makes no use of the value just input on link x. At the level of abstraction of our model, this value is not essential to the communication behavior of $A1$, but we assume that any input values are available for internal processing. In the π-calculus definitions given below, heavy use is made of the match operator to choose between alternative ways for an agent to proceed. This allows us to forego describing internal processing that relates input to output at the expense, however, of defining more processes.

In the definitions given below, to avoid additional commentary, a description (e.g., "anomaly resolution procedures") is sometimes given in the position of an output value in a negative prefix. For simplicity, only a small number of representative values of the parameters of the system are accommodated. Also, we occasionally use a negative prefix without an output value (e.g., $\overline{y}$ instead of $\overline{y}x$) and a corresponding positive prefix without a bound variable (e.g., y instead of $y(z)$. It is trivial to make the cases conform to strict monadic calculus syntax, but doing so would introduce names with no obvious purpose. Finally, the definitions overspecify the system in that, given two values to be communicated in any order, the linear order of nested prefixes in a process expression requires us to settle on one order or the other. This nuisance (like the previous inconvenience) could be avoided by using the polyadic version of the π-calculus.

The following, then, are the π-calculus definitions for modeling a system that supports the scenario in question. The mapping of scenario steps to definitions supporting those steps is given first:

A $\mapsto$ 1-2, B $\mapsto$ 3-5, C $\mapsto$ 6-8, D $\mapsto$ 9-10, E $\mapsto$ 11-12,
F $\mapsto$ 13-14, G $\mapsto$ 15-16, H $\mapsto$ 17-19, I-J $\mapsto$ 20-21, K $\mapsto$ 22-23

$$GIFA =_{def} \overline{amsg}(124000).GIFA1 + \overline{amsg}(246000).GIFA1 \quad (1)$$

$$FIRE =_{def} amsg(x).([x = 124000].FIRE1 + [x = 246].FIRE2) \quad (2)$$

$$FIRE1 =_{def} \overline{mvreq}(tm12st).\overline{mvreq}(tm13st).FIRE3 \quad (3)$$

$$FIRE2 =_{def} \overline{mvreq}(tm14st).\overline{mvreq}(tm15st).FIRE3 \quad (4)$$

$$GIFA1 =_{def} mvreq(x).mvreq(y).([x = tm12st][y = tm13st]GIFA2 + [x = tm14st][y = tm15st]GIFA3) \quad (5)$$

$$GIFA2 =_{def} \overline{amsg}(45).\overline{amsg}(47).GIFA \quad (6)$$

$$GIFA3 =_{def} \overline{amsg}(50).\overline{amsg}(52).GIFA \quad (7)$$

$$FIRE3 =_{def} amsg(x).amsg(y).FIRE4 \quad (8)$$

$$FIRE4 =_{def} \overline{areq}(\text{anomaly resolution procedures}).FIRE5 \quad (9)$$

$$\begin{aligned}
DBIFA &=_{def} areq(x).DBIFA1 && (10)\\
DBIFA1 &=_{def} \overline{matproc}(\text{fault resolution procedures}).DBIFA \\
&\quad + \overline{matproc}(\text{no fault resolution procedures}). \\
&\qquad DBIFA && (11)\\
FIRE5 &=_{def} matproc(x). \\
&\quad ([x = \text{fault resolution procedures}]FIRE \\
&\quad + [x = \text{fault resolution procedures}]FIRE6) && (12)\\
FIRE6 &=_{def} \overline{humreq}.FIRE7 && (13)\\
UIFA &=_{def} humreq.UIFA1 && (14)\\
UIFA1 &=_{def} \overline{humres}(\text{no user}).UIFA2 \\
&\quad + \overline{humres}(\text{user}).\overline{humres}(\text{resolve fault}).UIFA && (15)\\
FIRE7 &=_{def} humres(resp). \\
&\quad ([resp = \text{user}]humres(\text{resolve fault}).FIRE \\
&\quad + [resp = \text{no user}]FIRE8) && (16)\\
FIRE8 &=_{def} \overline{humreq}(sighum).\overline{humreq}(respsig). \\
&\quad \overline{reqhum}(sighum).\overline{reqhum}(respsig).FIRE9 && (17)\\
UIFA2 &=_{def} humreq(x).humreq(y).UIFA3(x,y) && (18)\\
PAGER &=_{def} reqhum(x).reqhum(y).PAGER1(x,y) && (19)\\
PAGER1(x,y) &=_{def} \overline{x}.y.PAGER && (20)\\
UIFA3(x,y) &=_{def} x.\overline{y}.UIFA4 && (21)\\
UIFA4 &=_{def} \overline{humres}(\text{user}).\overline{humres}(\text{resolve fault}).UIFA && (22)\\
FIRE9 &=_{def} humres(resp1).humres(resp2).FIRE && (23)
\end{aligned}$$

For process identifiers, we have used the names of agents, possibly followed by a positive integer. The right-hand side of each definition consists of a summation (allowing the case where no '+' combinator occurs to be the trivial, one-summand case) where each summand has at least one prefix preceding a single process identifier. In any given definition, the process identifiers on the right-hand side use the same name as the one of the left but use numbers greater than it in sequence (taking an unadorned agent name to contain a virtual '0'). The set of definitions, therefore, is partitioned into five subsets, one for each agent in the flowgraph. Each subset defines a transition diagram for the corresponding agent, shown in Figure 2, where the number of the definition of each process is shown to the left of its identifier. The actions labeling the transitions have been omitted. For the transition from process Xm to process Xn, the action is the string of prefixes preceding Xn in the definition of Xm. Other branches of the scenario (and other anomaly-system scenarios) would be accommodated by including additional paths in the transition diagrams. Embedding the anomaly system into the entire LOGOS community would require further elaboration. It is critical that the transition diagrams are coordinated by handshake communication.

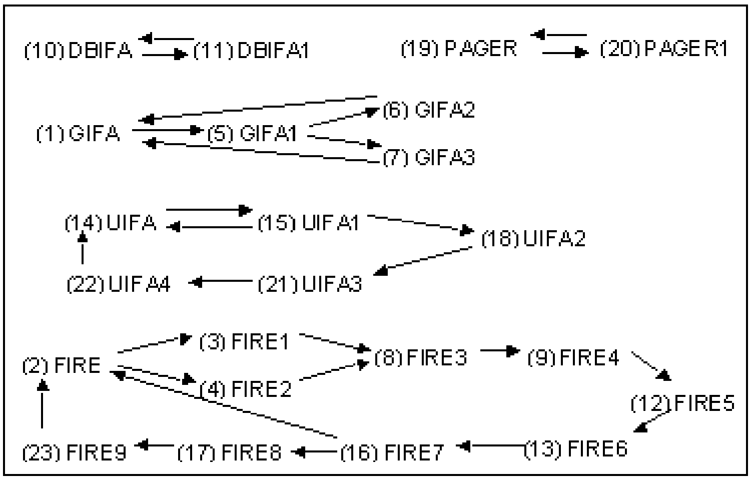


Fig. 2. Process Transitions

The above process definitions are easy to follow given our discussion. Note that definitions 17-19 capture the behavior whereby FIRE communicates the links *sighum* and *respsig* first to UIFA (on link *humreq*) then to PAGER (on link *reqhum*). Thereafter, variables x and y in both $UIFA3(x, y)$ and $PAGER1(x, y)$ are bound to *signhum* and *respsig*, respectively. Definitions 20 and 21 show PAGER outputting on a link and inputting on another and UIFA inputting on a link then outputting on another. The variable name x is used in both first actions, and y is used in both second actions. There is no significance to variable names agreeing across definitions. What is critical is that the agents are coordinated so that the handshakes allowed by definitions 17-19 indeed occur so that UIFA and PAGER may perform the intended handshakes. Although this specification allows only one binding for these variables, it is easy to enhance the it so that several alternative pairs of links may be passed. Although the prefixes in definitions 20 and 21 do not strictly conform to the monadic calculus since no values are passed, this can be trivially corrected.

5 Analysis and Design

We distinguish two processes if and only if the distinction is detectable by an agent interacting with both. If two processes cannot be distinguished in a given sense, then they are equivalent in that sense. Various equivalence relations are defined in the π-calculus (as in CCS) in terms of various notions of *bisimulation*: two equivalent processes must be able to perform the same actions, and, when they perform the same action, they must yield equivalent processes. The theory of bisimulations has been developed mostly in the polyadic version of the π-calculus, where they are defined over *agents* (or *p-agents*, to avoid confusion), which include not only processes but also *abstractions* (roughly, processes expecting an input, where the link in the prefix has been factored out) and *concretions* (similarly factored processes expecting output). As with CCS, bisimulations differ on whether they take silent actions into account, and some π-calculus bisimulations differ on how they are sensitive to substitutions for parameters. There are also equational laws based on congruences that support equational reasoning with process expressions (replacing equals with equals within an expression) for both CCS and the π-calculus. (A given equivalence relation $\equiv$ is a congruence relation when, if $A \equiv B$, then $C \equiv C[B/A]$ for any term C, where $C[B/A]$ is the result of replacing A in C with B.) The major bisimulation relations for the π-calculus are near congruences, failing to be closed only under abstraction.

A development methodology supported by these notions is as follows. Given a specification of an agent or system expressed as a π-calculus term, we produce a sequence of refinements, each provably equivalent to its predecessor, so that the final design is shown to be equivalent to the original specification. If the equivalence relation used is a congruence, then this approach supports modular design, for then a component of the specification can be refined and the result substituted for the original version within the overall specification, giving an expression provably equivalent to that specification. Equational laws based on congruences also support modular design, and they allow us to reduce a proposed design to the specification, thereby establishing equivalence.

The Hennessy-Milner logic $\mathcal{L}$ for the π-calculus is similar to the logic $\mathcal{PL}$ for CCS, but it must accommodate abstractions and concretions in general, not just processes. $\mathcal{L}$ has the usual sentential connectives, the two Boolean constants, and several modal operators. Where A is a p-agent and φ is a formula of $\mathcal{L}$, $A \models \varphi$ means that A *satisfies* φ (or φ is true of A). $\mathcal{L}$ is an alternative to the bisimulation relations since the aim is to define the satisfaction relation so that p-agents A_1 and A_2 are bisimilar exactly when $A_1 \models \varphi$ if and only if $A_2 \models \varphi$ for all formulae φ of $\mathcal{L}$. Where α is an action, $P \models <\alpha> \varphi$ holds if it is possible for P to do α and thereby reach a state satisfying φ. As $<\alpha>$ is a possibility operator (like the diamond operator of dynamic logic), its dual, $[\alpha]$, is a necessity operator (like the box operator of dynamic logic): $P \models [\alpha]\varphi$ means that, if P can perform α, it must thereby reach a state where φ holds. The logic $\mathcal{L}$ allows us to begin the development cycle, not with a complete specification of a system, but with only certain conditions on its behavior. The set of conditions can be augmented incrementally and applies throughout the refinement process.

The Mobility Workbench (MWB) [19] was developed in the pattern of the Concurrency Workbench (CWB) [5] but addresses the polyadic π-calculus instead of CCS. Besides deciding equivalence, it supports some of the other techniques of the CWB, such as finding deadlocks, and it supports interactive simulation. The MWB can also determine whether an agent satisfies a formula, using the logic $\mathcal{L}$ supplemented (as $\mathcal{PL}$ is supplemented in the CWB) with the greatest and least fixed-point operators of the μ-calculus. These operators allow familiar operators from branching-time temporal logic to be encoded and hence are particularly useful for formulating safety and liveness properties.

A version of the anomaly system without the mobility aspect was translated into the notation used by the value-passing CCS translator [3]. The CWB does not support value passing (input and output prefixes cannot take parameters). The value-passing translator translates value-passing CCS expressions into equivalent sets of non-value-passing expressions that can be input to the CWB. Since the CWB is considerably more mature than the MWB, this allowed us conveniently to explore aspects of the specification (without mobility) and refinements of it before moving to the MWB. The anomaly system including the mobility aspect was translated into the MWB, which accommodates value passing, and similar properties were explored.

6 Modeling Multiagent Systems in General

A framework should exploit the major conceptualizations that structure its domain. We briefly look at two conceptualizations that are nearly constitutive of the domain of multiagent systems: agent architectures and agent communication languages.

6.1 Agent Architectures and the π-Calculus

In [12] it is shown how to translate the λ-calculus into the polyadic π-calculus. Thus, by Church's Thesis, a multiagent system can be simulated in the polyadic π-calculus. Imagine the π-calculus supplying the model for programming-language constructs as the λ-calculus has supplied the model for constructs used in functional programming languages. Lists are available using an encoding like that used in the λ-calculus. Action-continuation patterns, like those heavily used in AI programming, are captured by prefix and summation. Link passing, which, among other things, allows us to simulate procedure call, allows modularity. Restriction allows us to internalize links within an agent process so that its actions are unobservable from outside and may correspond to internal computations. Given such constructs that allow us to avoid low-level details, it remains to determine, however, which aspects of multiagent systems may be fruitfully modeled in the π-calculus. We here sketch a few suggestions relative to BDI (Beliefs, Desires, and Intentions) architectures, which lie between purely reactive and deliberative planning systems and are the best articulated and motivated architectures for multiagent systems. BDI architectures attribute to systems attitudes falling into three categories. *Cognitive* attitudes include epistemic attitudes such as belief and knowledge, *conative* attitudes relate to action and control and include intention and having plans, and *affective* attitudes refer to the motivations of an agent and include desires and goals. (While both desires and intentions are pro-attitudes, only intentions involve *commitment*, which influences future activity and deliberation.)

An influential early, single-agent BDI architecture was IRMA [1]. It introduced stores for beliefs, desires, adopted plans ("intention structured into plans"), and plans as recipes (beliefs about what plan is applicable for accomplishing which task under what circumstances). Planning here involves filling in the gaps in skeletal plans and composing plans. In the π-calculus, a sequential plan could be represented by a sequence of actions (using prefix), and branches in a conditional plan could be captured by the combination of prefix and summation. An action not generally thought of as a communication (e.g., seizing an object or moving to a location) can be represented as a communication with the environment. The composition of plans in the π-calculus is obvious. For filling out skeletal plans, the position for a subplan could be occupied by an action that passes a link to a process responsible for the subplan. Updatable stores could be accessed by link passing for modularity. Plans as recipes are alternatives not yet realized, and adopted plans are already selected alternatives. Beliefs and desires could be represented as lists. The IRMA processes that operate on the stores require considerable flexibility. Still, implementations of such processes make heavy use of action-continuation patterns.

An influential BDI architecture addressing multiagent systems is GRATE* [8], which introduces joint intention in collaborative activity. If an agent decides to pursue an objective collaboratively, then initial team selection results in a skeletal joint intention. A second phase finalizes the outline of the proposal and the assignment of actions to individuals. Each group member must fit its assigned action in with its existing commitments. The challenge here is modeling what goes before finalization of a joint intention. Initial team selection could be modeled with communication patterns that poll agents and action patterns that structure groups of agents. Role assignment could be modeled by communication patterns that culminate in passing links that enable various roles. Finalizing the structure of the joint intention could be modeled as we suggested for filling out skeletal plans. Although several agents are typically involved here, communication and action patterns might be used to model negotiation and decision.

6.2 Semantics for Agent Communication Languages

We advocate the standard position of characterizing an agent communication language (ACL) in terms of performatives (bearers of the illocutionary forces of speech acts) and what may be done with them. The core issues in the semantics of ACLs then hinge on the semantics of speech acts, which Singh [16] presents by formally describing the conditions of satisfaction for the different types of speech acts. Singh's semantics relies on the notion of whole-hearted satisfaction of a speech act. For example, the whole-hearted satisfaction of a directive requires that an agent to whom it is addressed forces the appropriate state of affairs to come about, which in turn requires that the agent know how to bring about this state and intend to bring it about. The language used is propositional branching-time logic CTL* augmented with operators for intention and know-how. Singh's semantics, therefore, relies heavily on dispositions of the interlocutors, and the logic favors propositions over actions.

The modal logic $\mathcal{L}$ is from a different tradition—Hennessy-Milner logics—where modalities are formed from sequences of actions. We suggest that $\mathcal{L}$ enhanced with the greatest and least fixed-point operators of the μ-calculus (as in the MWB) could be used to represent the semantics of speech acts, complimented with the π-calculus process expressions. This would relate the meanings of performatives to patterns of actions and reduce the reliance on the dispositions of the interlocutors. (In linguistics, Fujinami [6] has used the π-calculus in performance theory to model flow of information with a system of communicating processes.) Still, we admit the need for whole-hearted satisfaction because the nondeterminism of π-calculus processes puts no limit on when handshakes must happen and generally is indifferent to several alternative actions.

7 Conclusion and Future Work

We have presented a formal framework that uses the π-calculus for modeling multiagent systems. This framework explicates the agent abstraction as a π-calculus process that persists through communication actions. Our principal task has been to illustrate the use of this framework by modeling certain aspects in the specification of NASA's LOGOS agent community. We have also sketched how a π-calculus framework supports development activities. Generally, we need more experience exploiting formal methods within our framework and in using tools that facilitate the associated activities. These methods must be specialized to multiagent systems, possibly via notions from particular agent architectures. We have suggested how certain architectural features may be expressed in our framework, but this is just a beginning. A π-calculus semantics for an agent communication language, as discussed in this paper, could help and should be significantly more tractable. The strength of a process algebra clearly lies with its handling of communication, but concurrency and communication are some of the most essential features of multiagent systems, as witness wrapping legacy systems in agent interfaces.

The proposed framework perhaps has a bias toward interfaces and communication. This bias, however, is shared by much work over the last few decades in distributed systems, human-computer interaction, linguistics, and philosophy, as witness such notions as those of a speech act, a situation, the flow of information, and common knowledge (perhaps better called shared information). This paper has argued that the language and techniques of the π-calculus are appropriate for modeling multiagent systems, but more generally we can address humans and their artifacts, the actions of both, and the situations they are in. After all, a major insight of the field of multiagent systems (and of AI in general) is the interoperability of humans and their artifacts. We have applied modal logics to human-computer integration [4] and are extending our π-calculus framework in this direction and beyond, to ubiquitous computing.

References

1. Bratman, M., Israel, D.I., and Pollack, M.E., Plans and Resource-Bounded Practical Reasoning, *Computational Intelligence*, vol. 4 (1988), pp. 349-355.

2. Brown, B., *High-Level Petri Nets for Modeling Multi-Agent Systems*, MS project, Dept. of Comp. Sci., North Carolina A&T State Univ., Greensboro, NC, 1998.
3. Bruns, G., *Distributed Systems Analysis with CCS*, Englewood Cliffs, NJ: Prentice-Hall, 1997.
4. Burge, J. and Esterline, A.C., Using Modal Logics to Model Societies of Agents, *Proc. IC-AI'2000*, Las Vegas, NV, 2000.
5. Cleaveland, R., Parrow, J., and Steffan, B., The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems, *ACM TOPLAS*, vol. 1, no. 1 (Jan. 1993), pp. 36-76.
6. Fujinami, T., *A Process Algebraic Approach to Computational Linguistics*, Center for Cognitive Science, University of Edinburgh, 1996.
7. Huhns, M.N. and Stephens, L.M., Multiagent Systems and Societies of Agents, in G. Weiss (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Cambridge, MA: The MIT Press, 1999, pp. 79-120.
8. Jennings, N.R., Specification and Implementation of a Belief-Desire-Joint-Intention Architecture for Collaborative Probem Solving *Int. Journal of Cognitive Information Systems*, vol. 2, no. 3 (1993), pp. 289-318.
9. Liu, Y. and Esterline A.C., *Prima Facie* Obligations and a Deontic Transaction Model For Multiagent Systems, *Proc. IEEE SoutheastCon 2000*, Nashville, TN, April 2000.
10. LOGOS Development Team, Automation Technology Section, *LOGOS Requirements & Design Document*, Greenbelt, MD: NASA/GSFC, 1997.
11. Milner, R., *Communication and Concurrency*, New York: Prentice-Hall, 1989.
12. Milner, R. The Polyadic π-Calculus: a Tutorial, in F. L. Bauer, W. Braueer, and H. Schwichtenberg (eds.), *Logic and Algebra for Specification.* Berlin: Springer-Verlag, 1993, pp. 203-246.
13. Milner, R., *Communicating and Mobile Systems: The π-calculus*, Cambridge: Cambridge University Press, 1999.
14. Milner, R., Parrow, J., and Walker, D. A Calculus of Mobile Processes, Parts I and II. *Journal of Information and Computation*, Vol. 100, 1992, pp. 1-77.
15. Rash, J., *LOGOS FIRE Agent Concept & Design: Fault Isolation Resolution Expert Agent*, Greenbelt, MD: NASA/GSFC, 1997.
16. Singh, M. P., A Semantics for Speech Acts, in M. N. Huhns and M. P. Singh (eds.), *Readings in Software Agents.* San Francisco: Morgan Kaufmann, 1998, pp. 458-470.
17. Singh, M.P., Rao, A.S., and Georgeff, M.P., Formal Methods in DAI: Logic-Based Representation and Reasoning, in G. Weiss (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Cambridge, MA: The MIT Press, 1999, pp. 331-376.
18. Tretyakova, Y. and Esterline, A.C, The Logic of Action in the Deontic Transaction Model, *Proc. IEEE SoutheastCon 2000*, Nashville, TN, April 2000.
19. Victor, B. and Moller, F., *The Mobility Workbench—A Tool for the π-Calculus*, Tech. Rep. DoCS 94/45, Dept. of Comp. Sci., Uppsala Univ., Sweden, 1994.
20. Wooldridge, M., Intelligent Agents, in G. Weiss (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Cambridge, MA: The MIT Press, 1999, pp. 27-77.
21. Wu, X., Cox, B.D., and Esterline, A.C., Representing and Interpreting Multiagent Plans with Statecharts, *Proc. WAC 2000*, Maui, Hawaii, 2000.

Specifying and Checking Fault-Tolerant Agent-Based Protocols Using Maude

Jeffrey Van Baalen[1], James L. Caldwell[1], and Shivakant Mishra[2]

[1] Department of Computer Science, University of Wyoming, Laramie, WY 82071
[2] Department of Computer Science, University of Colorado, Boulder, CO 80309

1 Introduction

Fault tolerance is an important issue in mobile, agent-based computing systems. However, most research in this area has focused on security and mobility issues. The DaAgent (Dependable Agent) system [8] is similar to several other agent-based computing systems including Agent-Tcl [6], Messengers [5], and Ajanta [9]. However, unlike these systems DaAgent is being designed to address fault tolerance issues. Within the DaAgent system several fault tolerant protocols are being investigated. These protocols have been specified in natural language, English prose, and a Java implementation within the DaAgent system is being tested. This approach has proved to be extremely time-consuming and inflexible, for example, it is difficult to rapidly change test conditions and fault injection is extremely primitive involving physically halting or reseting a machine, pulling network connections, or sending a kill message to the agent process. Testing the implementation of course serves as a weak form of evidence of correctness, but offers little real assurance of the correctness of the system.

In order to address these problems we are taking the approach of formalizing the protocol specifications and debugging them at the specification level. We are using the Maude executable specification language [2] for this purpose. We are taking this approach knowing that a majority of defects in systems can be traced back to specification errors. Also, much data supports the fact that errors identified early in the development process are dramatically less expensive to correct.

However, while simply specifying systems formally often identifies errors and ambiguities early, formal specifications can also contain errors. We have chosen to use an executable specification language so that we can debug our specifications without necessarily resorting to formal proofs of correctness or massive testing efforts. We have found that using Maude to specify our protocols has a number of benefits:

1. The Maude system has clean extendable syntax so that it is straightforward to create domain specific languages with appropriate abstractions that have a clearly defined semantics. This is an often undervalued feature that is very useful in a specification language, making it more understandable for the intended user.

2. The resulting specifications are executable and, in fact, the Maude engine is very fast (up to 1.3 million rewrites per second on a Pentium II). This enables us to construct a variety of initial states and explore very extensively the execution of the protocol from those states.
3. Maude is reflective. This enables us to separate a specification from the execution of that specification. This has proved useful in three ways. First, it has enabled different explorations from different initial states without changing our protocol specifications. Second, it has enabled a more sophisticated model-checking analysis in which all behaviors from some initial state are explored (up to some depth) via one of many possible search strategies (e.g., breadth first). Third, it has enabled us to specify fault models for our protocols separately from the protocol specifications. This has enabled the testing of the protocols, while injecting different types of faults.

The major contribution of this work is the separate specification of fault models at the meta-level enabling testing of protocols under conditions where faults are injected. Also, the fault injection technique can be combined with the search strategies, enabling the exhaustive testing of a protocol, injecting faults in all possible ways.

2 Rewriting Logic

The Maude language has a declarative semantics based on rewriting logic [2, 7]. Rewriting logic extends algebraic specification techniques to concurrent and reactive systems by providing a natural model of concurrency in which local concurrent transitions are specified as rewrite rules. The model is very flexible allowing both synchronous and asynchronous models of concurrency as well as a wide range of concurrent object systems. The flexibility of rewriting logic has enabled Maude's use for formalizing many different kinds of systems (see [4] for descriptions of a few).

In rewriting logic the state space of a concurrent system is formally specified as an algebraic data type by means of an equational specification consisting of a signature and a set of conditional equations. The equations in the specification of the system state define equivalence classes of terms over the signature. Concurrent transitions are specified as rewrite rules of the form $t \Rightarrow t'$, where t and t' are terms in the signature (normally) containing variables. Rules describe concurrent transitions because they specify how a system state matching t can change to a system state where the term matching t is replaced by t' (appropriately instantiated).

More formally (following [2]), a theory in rewriting logic is a pair $\mathcal{R} = ((\Omega, \Gamma), R)$, where (Ω, Γ) is an equational specification with signature Ω and equational axioms Γ. The equations of Γ define the algebraic structure of the theory, i.e., they define equivalence classes on terms with signature Ω. R is a collection of labelled rewrite rules that specify concurrent transitions that can occur in the system axiomatized by $\mathcal{R}$. The rules in R are applied *modulo* the equations in Γ.

The state space of a concurrent object system can be specified as an algebra by means of an equational theory (Ω, Γ). The concurrent state of such a system, often called a *configuration*, usually has the structure of a *multiset* (an unordered collection allowing duplicates) containing objects and messages. Therefore, we can view configurations as built up by a binary multi-set union operator, which can be represented with empty syntax (i.e., juxtaposition) as $\_\ \_ : \mathbf{Conf} \times \mathbf{Conf} \Rightarrow \mathbf{Conf}$. (Following the conventions of mix-fix notation, under-bars indicate argument positions.) The multi-set union operator $\_\ \_$ is declared to satisfy the laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages can then be specified as elements of sorts **Object** and **Msg** respectively and these sorts can be given as subsorts of **Conf** so that a single object or message is a singleton configuration.

An *object* is represented as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's name, C is its class, the a_i's are the objects *attribute identifiers*, and the v_i's are the corresponding attribute *values*. The set of an object's attribute-values pairs is formed by application of the binary operator **_,_** which is associative and commutative, so that the order of the attributes of an object is immaterial.

A message is also represented as a term. So, for example, given an operator **from_to_hello** : **Oid** $\times$ **Oid** $\rightarrow$ **Msg** and two Oids (object identifiers) **A** and **B**, the term **(from A to B hello)** is a message.

Rules specify local transitions by describing how one configuration can be transformed into another. In a concurrent object system, a rule might transform a configuration containing an object and a message for that object into a configuration in which the message has been removed and the object has been updated to reflect receipt of the message. Such a rule would have roughly the form:

```
rl[name] :
  ((from A to D some-message) ⟨ D : Agent | some-attributes ⟩ Conf)
  ⇒ (⟨ D : Agent | updated-attributes ⟩ Conf) .
```

where A and D are variables of sort f **Oid** and **Conf** is a variable of sort Configuration. The left hand side of this rule matches any configuration containing an object named D and a message to that object. The variable Conf is bound to the other objects and messages in the matched configuration. The right hand side of the rule specifies the transformed configuration which differs from the original because the message has been removed and agent D's attributes have been updated to reflect the receipt of the message. The rest of the configuration (Conf) is unchanged by this rule.

More generally, a rule can specify any combination of messages and objects in its left and right hand sides and can specify a condition that the instantiated left hand side must meet for the rule to be applicable [4].

$$\begin{aligned} r(\boldsymbol{x}) \;:\; & M_1, \ldots, M_n, \langle O_1 : F_1 \mid atts_1 \rangle, \ldots, \langle O_m : F_m \mid atts_m \rangle \\ & \Rightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots, \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\ & \quad \langle Q_1 : D_1 \mid atts''_1 \rangle, \ldots, \langle Q_p : D_p \mid atts''_p \rangle \end{aligned}$$

$$M'_1, \ldots, M'_q$$
$$if\ C$$

where r is the rule's label, $\boldsymbol{x}$ is a list of the variables occurring in the rule, the M's are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition.

If two or more objects appear in the left hand side of a rule, the rule is *synchronous*. If there is only one object appearing on the left hand side, the rule is *asynchronous*. A concurrent object rewrite theory is called a *distributed object theory* if all of its rules are asynchronous [4].

Given a distributed object theory $\mathcal{R}$, rewriting logic provides an inference system [7] to deduce the finitary concurrent computations possible in $\mathcal{R}$. These computations are identified with proofs of the form $\alpha : C \longrightarrow C'$, where C and C' are terms representing system configurations.

In Maude, a rewriting logic theory $\mathcal{R} = ((\Omega, \Gamma), R)$ is specified as a module with the syntax:

```
mod module-name  is
        including module-list
        sorts sort-list
        subsort subsort-specs
       signatures
       variable-declarations
       equations
       rules
endm
```

where `sort-list` is a list of sorts to be used in the module, `subsort-specs` defines subset relationships between the sorts, `signatures` is a set of operator signature definitions, `variable-declarations` defines a set of symbols to be used as variables in the equations and rules of the module, `equations` is the set of equations in Γ, `rules` is the set of rewrite rules in R, and `module-list` is a list of module names whose rewrite theories get textually included in the module being defined. A simple example of a Maude module is

```
mod ND-INT  is
        including MACHINE-INT .
        sort NdInt .
        subsort MachineInt < NdInt .
        op _?_ : NdInt NdInt → NdInt  [assoc comm] .
        var N : MachineInt .
        var ND : NdInt .
        eq N  ? N = N .
        rl [choice]: N  ? ND ⇒ N .
endm
```

The module ND-INT defines a sort `NdInt` (nondeterministic integer) as a supersort of `MachineInt` (a builtin sort). It defines a commutative associative constructor of `NdInts` denoted by `?`. Its one equation serves to remove duplicate `MachineInts` from an `NdInt`. Because `?` is commutative and associative, this module's single rule chooses an arbitrary MachineInt from an NdInt.

An example use of `NdInt` is to rewrite an expression such as

```
(1 ? 5 ? 2 ? 1 ? 5) + (3 ? 11 ? 7 ? 3 ? 11)
```

which results in the sum of a `MachineInt` nondeterminisically chosen from the first `NdInt` and a `MachineInt` nondeterministically chosen from the second `NdInt`.

In addition to modules declared with the keyword `mod`, Maude supports other kinds of modules, including an object oriented module (**omod**) which we make use of in our protocol specification. **omod**s can declare *classes* and *subclasses.* Each class is declared with the syntax **class** $C \mid a_1 : S_1, \cdots, a_n : S_n$ where C is the class name and for each $a_i : S_i$, a_i is an attribute identifier and S_i is the sort of values for that attribute. Objects in a class are terms of sort **Object** and are written with the previously described syntax.

3 DaAgent in Maude

The DaAgent system runs on a network of processors. A DaAgent server runs on every node in the network that might be visited by an agent or from which an agent might be launched. A node in the DaAgent network consists of the composition of instances of three module types: an agent server, an agent consultant, and some number of agent watchdogs, one per mobile agent.

The DaAgent server on a node services requests from local clients on that node to launch their agents. In addition, it services an agent that migrates from some other node to that node. Every mobile agent is associated with an agent watchdog. Agent watchdogs oversee the functioning of their agents and control the migration of agents to other nodes.

Additionally, there is one agent consultant on every node. An agent consultant on a node determines if a new agent can be launched from that node or if an agent can migrate to that node. Consultants implement an admission control protocol to determine if new agents meet all of the security and computational requirements to run on their nodes.

One of the protocols we have formally specified is called the watchdog-controlled agent recovery protocol (WC-ARP). This protocol automates detection of node failure and the agent recovery process. The key idea is that it uses agent watchdogs on the nodes that an agent visited earlier to monitor the execution of the agent on the current node. The earlier agent watchdogs detect node failures and recover an agent in case of a failure.

In WC-ARP, each agent has an ordered set of the some fixed number (n) of the most recent earlier watchdogs an agent visited. This ordered set of earlier watchdogs is called the agent's *entourage.* An agent's current watchdog is (referred to as AW) and all members of the agents entourage ($AW, AW_1, \cdots, AW_n$) have associated weights ($W_0, W_1 \cdots, W_n$). These weights specify the priority of that watchdog for ensuring that the agent continues to execute. Since AW contains the most recently computed state of the agent it is the preferred watchdog to ensure the continued execution of the agent. So, in a typical assignment of weights, $W_i > W_j, 0 \leq i \leq n, i < j \leq n$.

During normal execution, AW sends an 'agentAlive' message to all members of the entourage periodically (every t time units). A watchdog AW_j concludes that $AW, AW_1, \cdots, AW_i$ have failed if, in the last $j * sp$ time units, it has not received an 'agentAlive' message from AW, nor has it received a 'migration support request' (msr) message from any of watchdogs $AW_1, \cdots, AW_i$. Here $sp = t * K$, with $K > 0$ being a protocol parameter. Hence, an agent watchdog concludes the failure of AW if it misses K consecutive 'agentAlive' messages.

AW may migrate the agent upon request or a watchdog AW_j may recover an agent if it believes that $AW, AW_1, \cdots, AW_{j-1}$ have failed. However, due to the possibility of a communication partition, AW_j can never be certain that another watchdog has failed.

A threshold ($th \in \mathbb{N}$) is a system parameter used to determine when an agent watchdog is authorized to migrate or recover the agent. A so-called runaway agent is a replicated instance of an agent incorrectly launched by an agent watchdog (say A_j) based on incorrect information that all agents $A_k, k > j$ have failed. If $th \geq (W_0+W_1+\cdots+W_n)/2$, runaway agents will never be launched, but in this case the protocol tolerates fewer faults. If $th < (W_0 + W_1 + \cdots + W_n)/2$, runaway agents may be created, but fault tolerance may be increased.

To migrate an agent, AW_j sends a 'migration support request' to every member of the agent's entourage. Then AW_j waits for 'migration supported' (ms) messages in reply. On receiving a 'migration support request', AW_i replies with a 'migration supported' message if it has not yet sent a 'migration supported' message or a 'migration support request' to any other agent watchdog. AW migrates the agent when it has received 'migration supported' messages from entourage members such that the sum of their W_i is greater than th.

An agent watchdog AW_j sends 'migration support request' messages to $AW_{j+1}, \cdots, AW_n$ when

1. AW_j concludes that $AW, AW_1, \cdots, AW_{j-1}$ have failed, and
2. AW_j has yet to send a 'migration supported' message to anyone.

Upon receiving a 'migration support request' message, a watchdog $AW_k, (k > j)$ replies with a 'migration supported' message if

1. AW_k concludes that $AW, AW_1, \cdots, AW_{j-1}$ have failed, and
2. AW_k has not yet sent a 'migration supported' or a 'migration support request' message to anyone.

AW_j recovers the agent from its local checkpoint when it has received 'migration supported' messages from entourage members such that the sum of their W_i is greater than th.

We formalize the WC-ARP protocol as an **omod** in Maude that contains classes for agents and agent controllers. These are defined as

```
class Agent-Controller |
     agents-pending : AgentSet, agents-migrating : Spairs,
     agents-fwd : Wtuples .
class Agent-Home .
class Agent-WatchDog | agents-running : AgentSet,
     awtg-support : Mpairs .
subclasses Agent-Home Agent-WatchDog < Agent-Controller .
class Agent |
     dsts : Principals, livTime : MachineInt, XTime : MachineInt,
     ent : Entourage .
```

The agent class models a mobile agent. Instances of this class have attributes dsts, which is a list of names of watchdogs to which the agent should be migrated, livTime, which specifies how long the agent should live on each watchdog, XTime, which is used to count how long an agent has been running on its current watchdog, and ent which is the agent's entourage.

The sort of the dsts attribute is an ordered list of Principals (a subsort of Oid) defined as

```
subsort Principal < Oid .
subsort Principal < Principals .
op none : -> Principals .
op _ _ : Principals Principals -> Principals [ assoc id: none] .
```

An entourage is an ordered i-tuple, ($i \leq n$) of agent watchdog names (Principals) defined as

```
subsort Principal < Entourage .
op emptyEnt : -> Entourage .
op _‘,_ : Entourage Entourage -> Entourage [ assoc id: emptyEnt] .
op addEnt : Principal Entourage -> Entourage .
```

The addEnt operator is used to add a new watchdog name to the front of an Entourage, ensuring that the entourage is a tuple of at most n principals. AddEnt is defines as

```
vars A B : Principal .
var Ent : Entourage .
eq addEnt(A,emptyEnt) = A .
eq  addEnt(A,(Ent,B)) =
  if (length((Ent,B)) == entSize) then (A,Ent) else (A,Ent,B) fi .
```

The agent controller class (and its subclasses) defines two kinds of agent controllers: agent watchdogs and agent homes. Agent homes have a set of pending agents (agents to be migrated), a set of agents that are in the process of migrating, and a set of agents that have been forwarded. Agent watchdogs have, in addition, a set of running agents and a set of agents awaiting support for migration. Agent homes do not require these two additional attributes because agents only begin on agent homes. Therefore, there is no need for an `agent-running` attribute, nor is there need for an `awtg-support` attribute because this is where agents are placed when a watchdog is awaiting 'migration supported' messages from the agent's entourage. When an agent starts out at its home, it has no entourage.

`Agents-pending` and `agents-running` are `AgentSets` which are unordered collections of `Agent` objects defined as

```
subsort Agent < Agents .
op `{_`} : Agents -> AgentSet .
op none : -> Agents .
op _`,_ : Agents Agents -> Agents [ assoc comm id: none ] .
```

The `agents-migrating` attribute is used to record, in an agent controller *AW*, the fact that a message has been sent to migrate an agent. Instances of the `agents-migrating` attribute are pairs of `Agents` and `MachineInts.` The second component of the pair is used to record the amount of time that has elapsed since the migration message was sent. If a 'migration accepted' message is not received after a fixed number of time units, *AW* concludes that the watchdog it attempted to migrate the agent to is unreachable.

We also define the different types of messages of the protocol

```
msgs from_to_migrate_ from_to_ma_
     : Principal Principal Object -> Msg .
msgs  from_to_agentAlive_ from_to_msr_ from_to_ms_
     : Principal Principal Principal -> Msg .
```

The 'agentAlive' message is the message that *AW* periodically sends to members of the agents entourage. To migrate an agent from controller A to watchdog B, A sends a 'migrate' message to B containing the agent. If B accepts the agent it sends an 'ma' message back to A. To obtain migration support for an agent `Ag`, a watchdog sends an 'msr' message to each member of `Ag`'s entourage. In response, entourage members send an 'ms' message to support an agent's migration.

The `awtg-support` attribute records information used in the protocol after *AW* sends 'msr' messages to an agent's entourage. The sort of values for this attribute is an unordered set of `Mpairs.` Each `Mpair` is an ordered pair consisting of an `Agent` and an ordered set of `Bpairs.` Each `Bpair` is a `Principal` paired with a boolean flag. When a watchdog sends 'msr' messages for an agent `Ag,` it places `Ag` along with a list of `Bpairs` for each watchdog in Ag's entourage in the value of the `awgt-support` attribute. The `Bpairs` are used to record whether or not an 'ms' message has been received from each member.

```
subsort Mpair < Mpairs .
subsort Bpair < Bpairs .
op _;_ : Principal Bool → Bpair .
op _;_ : Agent Bpairs → Mpair .
op none : → Bpairs . op none : → Mpairs .
op _‘,_ : Bpairs Bpairs → Bpairs [ assoc id: none ] .
op _‘,_ : Mpairs Mpairs → Mpairs [ assoc comm id: none ] .
```

The agents-fwd attribute of an agent controller is used to record the agents that the controller has forwarded. This is an unordered set of triples consisting of an agent, the number of hops the agent has taken since it was forwarded from this watchdog, and the number of time units since the last alive message was received from the forwarded agent.

The remainder of the WC-ARP **omod** is a collection of rewrite rules that specify the protocol as manipulations of the Configuration. Space does not allow the inclusion of the whole specification, but here are some examples. The following rule initiates the run of an agent. It matches an agent home that contains an agent in its agents-pending field. It places a migrate message into the Configuration and modifies the agent home, moving the agent from the agents-pending field to the agents-migrating field. It modifies the agent's entourage, initializes the migration timeout counter data structure, and increments the timeout counters (incAll of any other agents in the agents-migrating field

```
vars A B D : Principal .
var Dsts : Principals .
var N : MachineInt .
var Sp : Spairs .
var Agts : Agents .
rl [BeginRun] :
   (< A : Agent-Home | agents-pending :
                          { < B : Agent | dsts : (D Dsts),
                                          livTime : N,
                                          ent : Ent >, Agts },
                        agents-migrating : Sp >
     Conf)
   ⇒ ((from A to D migrate
            < B : Agent | dsts : Dsts, XTime : 0,
                          livTime : N, ent : addEnt(A, Ent) >)
        < A : Agent-Home | agents-pending :  { Agts },
                           agents-migrating :
                             (< B : Agent | dsts : Dsts,
                                            XTime : 0,
                                            livTime : N,
                                            ent : Ent > ; 0),
                           incAll(Sp) >
        Conf) .
```

When the `XTime = livTime` for an agent, the next rule sends 'migration support messages' to the agent's entourage (the operator `sendMsrs` constructs these), setting up the data structure (`mkBpairs`) to record 'migration supported' replies.

```
crl [requestMigrate] :
   (< A : Agent-WatchDog | agents-running :
                              { < Ag : Agent | XTime : N,
                                               livTime : N,
                                               dsts : Dsts,
                                               ent : Ent >, Agts},
                           agents-fwd : Wp,
                           awtg-support : Mp >
    Conf)
   ⇒ (sendMsrs(A,Ag,Ent)
      < A : Agent-WatchDog | agents-running : { Agts },
                             agents-fwd : incAll(Wp),
                             awtg-support :
                               ((< Ag : Agent | XTime : N,
                                                livTime : N,
                                                dsts : Dsts,
                                                ent : Ent > ;
                                  mkBpairs(Ent)), Mp) >
       Conf)
  if (Dsts =/= none) .
```

The formalization of WC-ARP as a theory in rewriting logic uncovered a number of inconsistencies and unspecified conditions in the English specification. For example, the English specification did not address the issue of what to do if a migration message is never replied to. Also it did not address the migration condition early in an agent's tour before its entourage contains n watchdogs.

Most importantly, the formal Maude specification is executable: the result of running the WC-ARP protocol on an initial configuration `initConf` is simulated by rewriting `initConf` in the WC-ARP theory. The resulting configuration (assuming termination) is the term of sort `Configuration` that results. We have simulated the protocol from a number of different initial situations and these simulations have exposed additional errors and omissions in the protocol specification.

We have also formalized, as additional rewrite rules, properties that we would like the WC-ARP protocol to have, for example, no duplicate agents are ever created if the threshold and weights have appropriate values. These rules record state information and halt the simulation if the properties are ever violated. Notice that these simulations do not test that the protocol has desired properties when faults occur. The real benefit of using Maude is realized when its reflective capabilities are exploited.

4 Using Reflection in Maude

Informally, a reflective logic is one in which aspects of its meta-theory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant meta-theoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself [2].

In Maude's formalization of reflection, terms are "quoted" to allow them to be manipulated at the meta-level (one step up the reflective hierarchy).[1] Maude provides operators to quote and unquote terms so that they can be moved up and down the reflective hierarchy. Modules are terms which can be manipulated just as all others terms. The operator **up :** `ModuleName` $\rightarrow$ `Term`, produces the meta-level representation of the module whose name is given as argument and the operator **up :** `ModuleName` $\times$ `Term` $\rightarrow$ `Term`, produces the meta-level representation of its second argument in the module whose name is given as its first argument. The meta-level representation of the `NdInt` module given in section 2 is:

```
up(NdInt) =
mod 'ND-INT is
    including 'MACHINE-INT .
    sorts 'NdInt .
    subsort 'MachineInt < 'NdInt .
    op '_?_ : 'NdInt 'NdInt → 'NdInt [ assoc comm ] .
    var 'N : 'MachineInt .
    var 'ND : 'NdInt .
    eq '_?_ [ 'N , 'N ] = 'N .
    rl [ 'choice ] : '_?_ [ 'N , 'ND ] ⇒ 'N .
endm .
```

Maude also enables meta-computation via the builtin operator **meta-rewrite** which takes three arguments: the meta-level representation of a module M, the meta-level representation of a term t, and an integer n. Rewriting a meta-rewrite expression of the form **meta-rewrite(**M,t,n**)** produces the meta-level representation of the term that results from performing n rewrites of the term **down(**M,t**)** in the module **down(**M**)**. In the case where n is 0, an unbounded number of rewrites are performed, halting when no rewrite rule applies, or looping forever if some rule is always enabled. For example, rewriting the expression:

```
 meta-rewrite(
    up(NdInt),
    up(NdInt,(1 ? 5 ? 2 ? 1 ? 5) + (3 ? 11 ? 7 ? 3 ? 11)),
   0)
```

results in `({'4} 'MachineInt) =` **up**`(4)`.

[1] For details of the quoting process, see [2]

A common use of meta-computation, discussed extensively in [3,1,2], is to explicitly specify a rewriting strategy. Maude rewrite theories are required to be neither terminating nor Church-Rosser. Strategies are used to control evaluation in such rewrite theories. It is possible to define a very wide variety of rewriting strategies using rewrite rules at the meta-level and then to execute a specification with one or more of these strategies.

In [4], this strategy mechanism is used to explore (in a breadth-first manner) all the possible rewrite sequences in a theory beginning with some initial state. Using this approach they were able to validate protocol specifications. We have adopted a similar meta-level rewrite strategy to explore the possible execution sequences of the WC-ARP protocol, also in a breadth-first manner.

5 Fault Models

We use the reflective facilities in Maude to inject faults into WC-ARP simulations without changing the WC-ARP theory. This has enabled us to validate the WC-ARP theory under different fault models. The fault modeling technique can be used in conjunction with the breadth-first search strategy to inject faults into a simulation at any desired point of the computation.

A basic strategy for injecting faults is to randomly interrupt a rewrite sequence in WC-ARP and modify the configuration being operated on to emulate the fault. For example, to inject a crash of a watchdog *AW* into a WC-ARP simulation, we interrupt the simulation at some random point and remove from the configuration *AW* and all messages to *AW*.

Randomly interrupting and modifying a rewrite sequence is straightforward to do at the meta-level. As a simple example, here is the partial specification of a meta-level module that runs a WC-ARP simulation for a random number of steps, removes a watchdog from the configuration at that point, and then continues the simulation to completion.

```
(omod RUN-WC-ARP is
        including META-LEVEL[WC-ARP] .
        op oneCrash : Term Qid → Term .
        op remWD : Term Qid → Term .
        op rand : MachineInt → MachineInt .
        var T : Term .
        var W : Qid .
        eq oneCrash(T,W) =
           meta-rewrite(WC-ARP,
                        remWD(meta-rewrite(WC-ARP,T,rand(seed)),W),
                        0) .
endom)
```

The operator `oneCrash` meta-rewrites a random number of times the meta-level representation of a configuration (`T`), removes the watchdog named `W` from the resulting configuration (`remWD`), and meta-rewrites the modified configuration until termination.

Clearly more general manipulations at the meta-level can be used to emulate many different classes of faults. For example, we can also simulate a watchdog crashing during a WC-ARP run and then recovering at some later point in the run by (1) meta-rewriting an initial configuration a random number of times, (2) removing the watchdog, (3) meta-rewriting the resulting configuration another random number of times, (4) adding the watchdog back into the configuration, (5) continuing meta-rewriting to completion.

Similarly, we simulate a communication partition by (1) meta-rewriting an initial configuration a random number of times, (2) removing any messages between watchdogs in separate partitions, (3) partitioning the watchdogs in the resulting configuration into two or more new configurations, (4) meta-rewriting these new configurations separately for a random number of steps, (5) then merging the configurations and meta-rewriting the merged configuration to termination.

So far we have modeled crashes, crashes with later recovery, and communication partitions as just described.

If, rather than generating a random number of meta-rewrite as described above, we parameterize the number steps, we can choose a set of values for these parameters and then apply the same fault scenario to all possible rewrite sequences (up to some depth) from an initial state. This enables model checking a simulation under a fixed fault scenario.

Note that the module implementing the breadth-first search executes at the meta-level with respect to the WC-ARP module, a module implementing a fault scenario must execute at the meta-meta-level with respect to the WC-ARP module. Also note that by adding one additional level to this reflective tower it is possible to vary the parameters controlling the number of steps in each part of a fault scenario. This enables the systematic checking of fault scenarios in any desired combination of rewrites and faults.

6 Conclusions and Future Work

We have proto-typed the fault-tolerant protocols used in the DaAgent system by formally specifying them as systems of rules in Maude's rewrite logic. Maude has been used to model check the protocol, including fault-tolerant aspects, by explicit state methods; this has been accomplished by using the powerful reflective capabilities provided by Maude. The use of reflection to separate the theory specifying the protocol from the execution and fault models is a powerful abstraction technique that offers what we believe is an unprecedented level of flexibility.

Currently, we are using Maude further explore the fault-tolerant protocols used in the DaAgent system.

In future work, we intend to extend our methods to include support for symbolic model checking. Maude also includes an inductive prover which we may apply to the problem of formally verifying certain properties of the protocols.

References

1. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, , and J. Meseguer. Metalevel computation in Maude. In C. Kirchner, , and H. Kirchner, editors, *2nd Intl. Workshop on Rewriting logic and its Applications*, volume 15. Elsevier, 1998.
2. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* Technical report, SRI International, Menlo Park, CA, Jan 1999.
3. M. Clavel and J. Meseguer. Reflection in rewriting logic and its applications in the Maude language. In *Proceedings of IMSA-97*, pages 128–139, Japan, 1997. Information-Technology Promotion Agency.
4. G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 251–265, Hilton Head, South Carolina, Jan 2000. IEEE Computer Society Press.
5. M. Dillencourt, L. F. Bic, and M. Fukuda. Distributed computing using autonomous agents. *IEEE Computer*, 28(8), Aug 1996.
6. R. S. Gray. Agent tcl: A transportable agent system. Technical report, Dartmouth College, November 1995.
7. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *Proceedings WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer Verlag, 1998.
8. S. Mishra, Y. Huang, and H. Kuntur. Daagent: A dependable mobile agent system (fastabstract). In *Proceedings of the 29th International Symposium on Fault-tolerant Computing*, Madison, WI, June 1999. IEEE.
9. A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R Singh. Mobile agent programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, TX, 1999.

Agents Talking Faster

Tim Menzies[1], Bojan Cukic[1,2], and Harhsinder Singh[3]

[1] NASA/WVU Software Research Lab, 100 University Drive, Fairmont, USA
[2] Department of Computer Science and Electrical Engineering, West Virginia University
[3] Department of Statistics, West Virginia University, Morgantown tim@menzies.com, cukic@csee.wvu.edu hsingh@stat.wvu.edu

Abstract. Studies of human conversation suggest that agents whose world models are in consensus can work well together using only very narrow bandwidths. The total bandwidth required between agents could hence be minimized if we could recognize when model consensus breaks down. At the breakdown point, the communication policy could switch from some usual-case low value to a temporary high value while the model conflict is resolved. To effectively recognize the breakdown point, we need tools that recognize model conflicts without requiring extensive bandwidth. A mathematical model of probing and-or graphs suggests that, for a large range of interesting models, the number of probes required to detect consensus breakdown is quite low.

1 Introduction

Agents running on distributed computers may interact under a variety of resource restrictions. One restriction is bandwidth. Every transmission also takes time since it must be processed by both the sender and the receiver. Studies of human communication suggests that failure to optimize the use of available resources can result in inefficient communication within organizations and consequent task errors [2]. How can we minimize the use of those resources for agents, and maximize the likelihood that communication tasks are successfully completed?

Consider two humans communicating via a channel with a certain bandwidth. A robust finding from human conversations is that restricting bandwidth has little impact on the effectiveness of the outcomes of many collaborative problem solving tasks [10]. This occurs when the communicating individuals have a high degree of shared common ground. A particularly striking finding is that humans assume they share common world views in conversation until they detect an error, and then attempt to probe the cause of the error. Interestingly, human conversation seems to consist of both an explicit small set of exchanges devoted to testing or confirming understanding, as well as the primary exchange [7].

Can we use these observations to minimize the bandwidth required between computational agents? Suppose agents move through alternating stages of assumed model consensus, detection of model conflict, and then model repair.

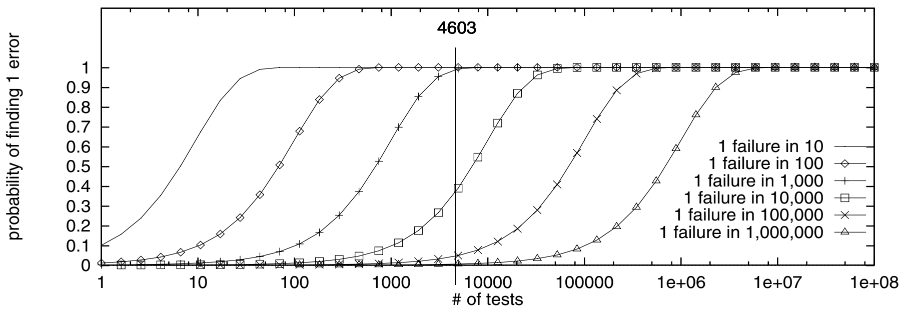


Fig. 1. Odds of finding an event with probability x after N trials.

While agent models are in conflict, we broaden the bandwidth between them. When models are in consensus, we restrict the bandwidth by assuming shared knowledge between agents. As a result, bandwidth is conserved during conversation until it is explicitly demanded to reconcile the world models of conversing agents. Time is also conserved by minimizing unnecessary sharing of knowledge between agents [3,4].

This "assume-consensus" assumption has another advantage. Consider agents with a learning component. Such agents may update their world model. If they wish to co-ordinate with other agents, then they would need to reflect on the beliefs of their fellows. In the worst case, this means that each agent must maintain one belief set for every other agent in its community. This belief set should include what the other agent thinks about all the other agents. If that second-level belief set includes the original agents beliefs, an infinite regress may occur (e.g. "I think he thinks I think that he thinks that I think that.."). This "assume-consensus" model addresses the infinite regress problem. Agents in assumed consensus only need to store their own beliefs and one extra axiom; i.e. "if I believe that she believes what I do, then my beliefs equals her beliefs".

To apply this approach, agents have to continually test that other agents hold their beliefs. One method for doing this would be to ask agents to dump their belief sets to each other. We consider this approach impractical. Firstly, it incurs the penalty of the infinite regress, discussed above. Secondly, such belief-dumps could exhaust the available band-width. Lastly, agents may not wish to give other full access to their internal beliefs. For example, security issues may block an agent from one vendor accessing the beliefs of an agent from another vendor.

Without access to internal structures, how can one agent assess the contents of another? Software engineering has one answer to this question: black-box probing [6]. Agent-A could log its own behaviour to generate a library of input-output pairs. In doing so, Agent-A is using itself as a specification of the expected behaviour of Agent-B, assuming our two agents are in consensus. If Agent-A sees those outputs when Agent-B is presented with those inputs, then Agent A could infer that Agent B has the same beliefs as itself.

Unfortunately, black-box probes can be very bandwidth expensive. A randomly chosen input has odds x that it will stumble across some fault. Further, this input will miss that fault with odds $(1-x)$. If we conduct N random black-box probes, then the odds of a failure not occurring (thus not revealing the fault) is $(1-x)^N$. Hence the probability of finding a fault, hiding in a program in N random tests is Equation 1 (and the inverse is Equation 2):

$$p(x, N) = 1 - \left((1-x)^N\right) \tag{1}$$

$$N(p, x) = log(1-p)/log(1-x) \tag{2}$$

Equation 1 is plotted in Figure 1. We see that $N(0.99, 0.001) = 4603$; i.e. to be 99% certain that Agent-B has less than a tenth percent difference between its beliefs and those of Agent-A, then 4603 probes would be required . Equation 1 is hence very pessimistic on the possibility of Agent-A accurately assessing its consensus with Agent-B, without using large bandwidths for the probing.

We have further cause for pessimism. If agents use some type of AI algorithm, the agent may be indeterminate; i.e. the same probe at different times may produce different results. Hence, the number of probes required to sample all that indeterminate behaviour may be very large.

This paper will argue that such pessimism is not always appropriate. Figure 1 assumes no knowledge of the internal structure of our agents. If we commit to some view of structure, then estimates can be generated for the odds of finding differences between Agent-A and Agent-B. Further, that estimation process will comment on the effects of indeterminacy. We will show that, surprisingly, if we pretend that our indeterminate agent is determinate, we generate almost maximal information of the agent.

This new results give us optimism that the following method will reduce inter-agent bandwidth. Firstly, assume consensus and restrict bandwidth. Next, as part of the normal inter-agent dialogue, drop probe questions. If the number of probes required is very small, then these process will add little to the overall bandwidth. If conflicts are detected, increase the bandwidth between Agent-A and Agent-B to allow for discussions.

The rest of this paper is structured as follows. § 2 is the bulk of the paper and presents a mathematical model of the odds of reaching some random node in an and-or graph. § 3 describes what happened when we ran that model 100,000 times. § 4 discusses the implications of that observed behaviour. Our conclusion will be that, at least half the time, testing for consensus breakdown will not require excessive extra bandwidth.

Before beginning, we digress to offer the following caveats. This paper extends a mathematical model of average case reachability within an and-or graph [12, 13]. The second version of that model allowed the frequency of conjunctions in a theory to be random variable. This newer version allow agents to loop through parts of themselves at different times. Regardless of the improvements in the model, it is still an average case analysis. Such an average case analysis may miss rare but mission-threatening events. Hence, this model should not be used in mission-critical situations. Also, we caution that the psychological likelihood of this model is questionable and we should not use these results to make statements about how humans should co-ordinate (exception: in the special case where humans are collaborating to test software, these results could be used to optimize their testing efforts.).

2 And-Or Graph Reachability

Roughly speaking, probing is a process of finding a needle in a haystack. What are the odds of finding some random needle? To answer this question, we have to commit to some model of a program. The following model applies to any program which can be reduced to a directed and-or graph between concepts. A large class of problems can be so translated. We show examples below of how to express procedural and logic programs as and-or graphs.

In our first example, Figure 3 shows the and-or space within the PROMELA program of Figure 2. PROMELA is a block-structured language where the "`::`" operator represents indeterminant choice. PROMELA is supported by the SPIN model checker [8] and is widely used in the model checking community.

```
01 byte a=1;
02 byte b=1;
03 bit f=1;
04
05 active proctype A(){
06  do
07  :: f==1 -> if
08              ::a==1 -> a=2;
09              ::a==2 -> a=3;
10              ::a==3 -> f=0;
11                        a=1;
12              fi
13  od
14 }
15 active proctype B(){
16  do
17  :: f==0 -> if
18              ::b==1 -> b=2;
19              ::b==2 -> b=3;
20              ::b==3 -> f=1;
21                        b=1;
22             fi
23  od
24 }
```

Fig. 2. Sample PROMELA code

As much as possible, and-or graphs must include the semantics of the program being represented. For example, note that the bottom nodes (`and4`, `and8`) in Figure 3 have three parents. These nodes were generated from lines 11 and 21 of Figure 2. Within SPIN, we can only reach, for example, line 11 after the command `F=0` executes on line 10 and `F==1` executes on line 7. Hence, the trio of parents.

And-or graphs can be *explicit* or *implicit*. Explicit and-or graphs pre-enumerate the entire search space. Implicit and-or graphs can be extended at runtime. For example, observe that in Figure 3 a proof to $A = 1$ at height $j = 8$ is traversing a loop (from `and4` back to `A=1`). As we execute around a loop, we extend the implicit and-or graph by adding more edges (one new edge for each time we re-use an edge in that loop).

We can use and-or graphs to prove program properties:

- Specify desired program properties.
- Negate those properties.

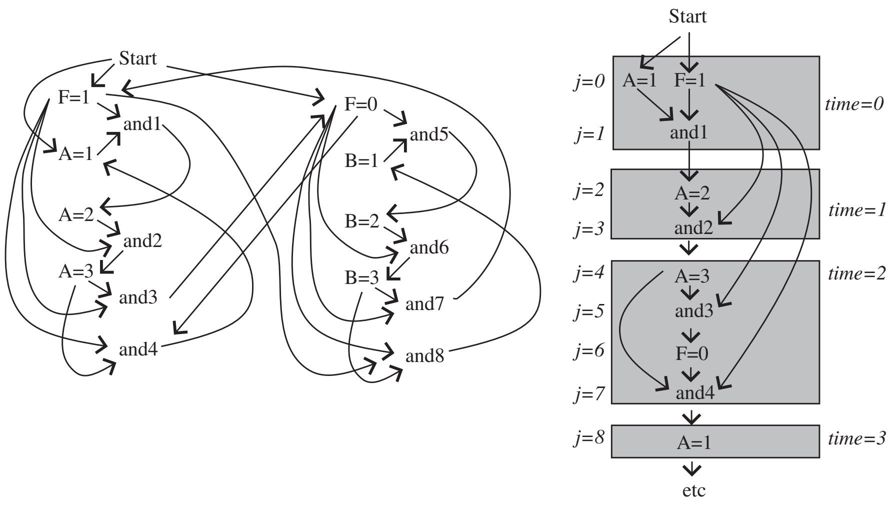


Fig. 3. The and-or graph within Figure 2. On the right, the shaded nodes show a pathway to a violation of the property `always not A=3` at height $j = 4$. Note that this path will take at least 3 time ticks to execute since it travels through `A=1, A=2`.

– Seek proof trees across the and-or graph to reach the negated properties.
– If such proof trees can be generated, then we have a counter-example that demonstrates how a program can fail.

Temporal logic model checkers (e.g. SPIN) take a similar approach. Constraints are expressed in temporal logic and the model checkers search for counter-examples. However, model checkers explore *state space*, not and-or graphs (for details, see [1]).

2.1 NAYO Graphs

A *NAYO* graph is a specialization of an and-or graph. NAYOs can be analyzed to find the average probability of reaching a node picked at random. NAYOs were first introduced in [12] and analyzed further in [13]. The following analysis extends our previous analysis since it deals properly with simulations that occur over multiple time ticks.

In a *NAYO* and-or graph, there are two types of nodes and two types of edges: n̲o-edges, a̲nd-nodes, y̲es-edges, and o̲r-nodes. As with and-or graphs, *or-nodes* store assignments of a single value to a variable and *and-nodes* model multiple pre-conditions. All the parents of an and-node must be reached before this node is visited. Only one of the parents of an or-node need be reached before we visit the or-node. Also, *yes-edges* model valid inferences while *no-edges* model incompatible inferences. Figure 4 shows a sample NAYO graph. Note the no-edge showing that we cannot believe in both a light and fatty diet.

No-edges slows our ability to process NAYO graphs. Building a path across a network with containing incompatible pairs can be mapped into 3SAT [5]; i.e. it is an NP-hard task with an exponential upper-bound on the runtimes. Since they can be so slow, NP-hard tasks are often implemented using heuristics. Such heuristics make (hopefully) informed guesses at runtime to negotiate a pathway across the graph. These heuristics are often implemented using some procedure mechanism and so may not be expressible as NAYO graphs. Hence, the choices that a inference engine will make at runtime across the NAYO graph cannot be determined by examining the graph. Since the pathways that will be taken cannot be determined, NAYO graphs can be indeterminate. Note that if a NAYO graphs lacks no-edges, then it need not be indeterminate.

```
happy            :- tranquillity( hi).
happy            :- rich , healthy.
healthy          :- diet(light).
satiated         :- diet(fatty).
tranquillity(hi) :- satiated.
tranquillity(hi) :- conscience(clear).
diet(fatty).
diet(light).
```

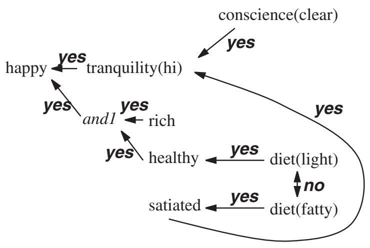


Fig. 4. Translating ground horn clauses (left) to a NAYO graph (right).

NAYO Graphs: Static Properties The NAYO graph contains a number of nodes denoted by V. Some fraction of these nodes are and-nodes ($andf$) and the rest are or-nodes (orf). Note that $orf + andf = 1$; i.e.

$$orf = 1 - andf \tag{3}$$

We model $andf$ as a the beta distribution $\beta(andf_\mu)$ with mean $0 \le andf_\mu \le 1$ We can execute the NAYO graph for T time ticks; e.g. Figure 3 shows an execution till $T = 3$. The effective size of the graph is therefore $T * V$.

In the NAYO graph:

– Or-nodes have orp number of parents (on average). We model orp as the gamma distribution $\gamma\left(orp_\alpha, \frac{orp_\mu}{orp_\alpha}\right)$ with mean $1 \le or_\mu \le \infty$ and "skew" orp_α[1].
– And-nodes have $andp$ number of parents (on average). We model $andp$ as the gamma distribution $\gamma\left(andp_\alpha, \frac{and_\mu}{and_\alpha}\right)$ with mean $2 \le andp_\mu \le \infty$ and skew $andp_\alpha$.

[1] In gamma distributions, low orp_α (e.g. $orp_\alpha = 2$) implies a highly skewed distribution while a high orp_α (e.g. $orp_\alpha = 18$) implies that the γ distribution approaches the normal distribution; i.e. is unskewed.

- Or-nodes contradict *no* number of other or-nodes (on average). We model *no* as the gamma distribution $\gamma\left(no_\alpha, \frac{no_\mu}{no_\alpha}\right)$ with mean $0 \leq no_\mu \leq \infty$ and skew no_α. Recalling the above discussion, we say determinant systems have $no_\mu = 0$ and a pre-condition for indeterminate systems is $no_\mu > 0$.
- No-edges only connect nodes which are found to be incompatible. Hence, and-nodes will never be touched by a no-edge.

Our view of testing comes from the model checking community; i.e. when we "test" an artifact, we seek a pathway to some violation of a property. This rest of this article constructs a probabilistic model of reaching an arbitrary part of a program represented as a NAYO graph.

NAYO Graphs: Dynamic Properties Our testing task is to build a pathway from some inputs to some problem state. This section describes some properties of a pathway from some inputs to a randomly selected node across a NAYO graph.

Each and-node in such a path will have at least one parent that is an or-node. We say that the root of that path is at height j, where j is the longest path from the root to any leaf (input). The size of the set of inputs is denoted by *in*. We declare that only or-nodes can be inputs. The probability P of reaching a random node at height 0 is hence

$$P[0]_{and} = 0 \tag{4}$$

$$P[0]_{or} = \frac{in}{V * T} \tag{5}$$

In the path, any node at height j ($j > 0$) will have one parent at height $j - 1$, and the rest at height i where $i < j$. The variable i controls how far back in the graph the node may have its parents: $i = \beta(depth) * (j - 1)$ where i is the beta distribution $\beta(depth)$ with mean *depth*. As *depth* decreases, parents come from further and further back in the NAYO graph. As we search the NAYO graph, at height j away from the inputs, we can characterize the local region in the graph as $andp[j], orp[j], no[j], andf[j]$.

The probability $P[j]_{and}$ of reaching an and-node at height $j > 0$ is the probability that all its parents are reached at height i ($i < j$):

$$P[j]_{and} = \prod_{1}^{andp[j]} P[i] \tag{6}$$

Also, or-nodes can be reached if at least one of their parents has been reached. That is, the probability $P[j]_{or}$ of reaching an or-node at height $j > 0$ is the probability that we will not miss all of its parents at height i, where $i < j$; i.e.

$$P[j]_{or} = 1 - \prod_{1}^{orp[j]} (1 - P[i]) \tag{7}$$

All non-input nodes have a number of immediate parents that is a weighted sum of the frequency of and-nodes and or-nodes, times the number of required preconditions. And-nodes require all their preconditions; ie. $andp[j]$ while or-nodes require only one of their preconditions. Hence, for $j > 0$, any node will have immediate parents $andf[j] * andp[j] + orf[j] * 1$. If no ancestors overlap from the found node back to the inputs, the the path is a tree. Every node in the tree must have all its parent in the tree as well; i.e. the number of nodes $n[j]$ in a tree of height j is:

$$n[j] = isTree? * \sum_{l=0}^{j} \left(\prod_{m=0}^{l} (andf[m] * andp[m] + orf[m]) \right) \tag{8}$$

$isTree?$ models our confidence in Equation 8. Equation 8 is clearly an upper-bound on $n[j]$. Firstly, the size of the path can never be bigger than the NAYO graph; i.e. $n[j] \le (V * T)$. Secondly, for small in, it is likely that the same nodes will be accessed multiple times; i.e. the path will not be a tree but a network with overlapping ancestors. We complete believe/reject the path-as-tree assumption when $isTree = 1$ or $isTree = 0$ respectively.

The probability $P[j]$ that a node can be reached at height j is the sum of $P[j]_{and}$ and $P[j]_{or}$, weighted by the frequency of *and* nodes and *or* nodes, i.e.,

$$P[j] = andf[j] * P[j]_{and} + orf[j] * P'[j]_{or} \tag{9}$$

$$P'[j]_{or} = P[j]_{or} \; * \; P[j]_{no\,loop} \; * \; P[j]_{no\,clash} \tag{10}$$

$P'[j]_{or}$ modifies $P[j]_{or}$ to check that our path is not in a loop, or is using incompatible nodes; i.e. uses nodes from both ends of a *no* edge. Recall that or-nodes can clash (contradict), on the average *no* other or-nodes. Clashes can only occur in the current time tick. Assuming that nodes are spread evenly across the T ticks in the simulation, the number of nodes that could clash with a node is:

$$clash[j] = \tfrac{n[j]}{T} * orf[j] \tag{11}$$

The probability $P[j]_{no\;clash}$ that a new node can be added to a NAYO path of size $n[j]$ at level j is the probability that this new node is not one of the nodes connected by no-edges to members of $clash[j]$:

$$P[j]_{no\,clash} = \left(1 - \frac{no[j]}{V}\right)^{clash[j]} \tag{12}$$

Not only must a new node not contradict with other nodes in the explanation tree, it must also not introduce a loop into the tree, since loops do not contribute to revealing unseen nodes. Hence:

$$P[j]_{no\,loop} = \left(1 - \frac{1}{V}\right)^{clash[j]} \tag{13}$$

From Equation 13 and Equation 12, we see that if we over-estimate $clash[j]$ we will under-estimate $P[j]_{or}$. Hence, we say that $isTree = 1$ gives a lower bound on $P[j]_{or}$ and $isTree = 0$ gives an upper bound on $P[j]_o r$.

We can summarize a run of the above model by reporting the average number of tests required to be be 99% confident that we can reach a node at any height:

$$P_{av} = \frac{\sum_{j=0}^{jMax} P[j]}{jMax} \tag{14}$$

$$N_{av} = N(0.99, P_{av}) \tag{15}$$

(where N comes from Equation 2). There are four interesting classifications of N_{av}:

Fast and cheap: Suppose a small number of randomly selected inputs can reach most nodes in the program representation graph. In this case, the overhead of manually reflecting over the design knowledge to construct a test set would clearly be useless.

Fast and moderately expensive: If a large, but not very large, number of randomly selected inputs can reach most nodes in the program representation graph, then testing is not "fast and cheap". However, it can still be afforded.

Slow and expensive: If a very large number of randomly selected inputs are required to reach most nodes in the program representation graph, then testing is not fast and may consume considerable resources to complete.

Impossible: If not the above three cases, then an impractically large number of tests are required to test a system. Note that this is an argument based on randomized testing. A careful manual selection of test cases might decrease the number of tests required to reach specific nodes in the graph. However, theoretical results suggest that such manual test case selection may not be significantly better [6].

3 Simulations

The above model was run 100,000 times as follows:

- To perform one simulation, we set our model's variables by picking one item at random from all the ranges shown in Figure 5 (top). We then calculated $P[i]$ for $j = 0, 1, 2, \ldots jMax$. During the run, we cache values for $P[j]$ and $n[j]$.
- We repeated the above procedure 100,000 times. Before each run we cleared the caches. After each run we used N_{av} to classify the results.

The results are summarized in Figure 5 (bottom). One surprise was that in a large number of cases (36%), less than 100 randomly selected inputs have a 99% chance of reaching any part of the graph. Further, ($36 + 20 = 56\%$) was either *fast and cheap* or *impossible*. Note that in that 56% of cases, a small number of randomly selected test cases will tell us as much as a very large number of tests. Further, in 23% of cases, it takes considerable effort to reach some part of the NAYO graph.

$$
\begin{aligned}
jMax &= 100\\
isTree?_\mu &\in 0, 1\\
V &\in 500, 1500, 2500, ..10^6\\
T &\in 1, 2, 3, ...10^2\\
in &\in 1, 6, 11, ...10^3\\
andp_\alpha, andp_\mu,&\\
orp_\alpha, no_\alpha &\in 2, 3, 4, ...18\\
orp_\mu &\in 1, 2, 3, 4, ...10\\
no_\mu &\in 0, 1, 2, 3, 4\\
depth, andf_\mu &\in 0.1, 0.2, 0.3, ...0.9
\end{aligned}
$$

Classification	Threshold	%
fast and cheap	$N_{av} < 10^2$	36
fast and moderately expensive	$N_{av} < 10^4$	19
slow and expensive	$N_{av} < 10^6$	23
impossible	$N_{av} \geq 10^6$	20

Fig. 5. Simulation inputs and outputs

The results shown in Figure 5 (bottom) do not tell us which of our parameters were most important. We can find this out by using the C4.5 decision tree learner [15]. C4.5 builds its trees via a information theory measurement of the entropy (information content) in each variable ranges. Low-entropy variable ranges will not appear in a learnt tree since they do not significantly effect the model's classifications. From the simulation runs generated above, C4.5 built a decision tree containing 10,739 nodes with an estimated error on unseen cases of 11.5%. To generate a tree we can view, we altered m: the sub-tree branch threshold. C4.5 continues to fork sub-trees until less than m examples fall into that sub-tree (default is $m = 2$). Figure 6 shows how m effects the size of decision trees learnt from our simulation data. Figure 7 shows a small tree generated at $m = 4096$ and Figure 8 shows a larger tree generated at $m = 512$. Note that as m increases and C4.5 explores fewer sub-trees, the trees become less detailed (i.e. they shrink) while the estimated error of the tree on unseen cases also increases.

By changing the m value, we can sort our model's parameters according to their information content. As m increases, the learnt trees shrink and the low-entropy variables disappear from the trees. Variable ranges with highest information content appear at the larger m values. Figure 6 shows that the highest information content variables were $andf_\mu$, orp_μ, and $depth$. That is, of all the variables in the simulation, the ones with greatest impact on the odds of reaching a node are the mean number of or-node parents, the frequency of and-nodes, and how far back into graph a node reaches for its parents.

The variable ranges with lowest information content appear at the smaller m values. One such low-information variable was $isTree?$: it did not appear till $m = 32$. We can hence ignore $isTree?$, at a cost of 6% more errors in our classifications (the different between the 18.5% error at $m = 128$ and the 11.5% error at $m = 2$). Note that is this 6% increase in errors is acceptable, we could approximate Equation 12 as just $P[j]_{no\ clash} = 1$.

Other variables with low information content are the indeterminacy controlling variables: no_α and no_μ. Repeating the above argument, if a 6% increase in errors is acceptable, we could ignore the no variables; i.e. do not distinguish between determinant and indeterminate systems.

	m	4096	2048	512	128	32	8	2
	Tree size	21	37	117	437	1311	3855	10739
	Estimated error	32.3%	28.6%	23.3%	18.5%	15.7%	13.4%	11.5%
	Displayed in ...	Figure 7		Figure 8				
entropy= very high	$andf_{\mu}$	●	●	●	●	●	●	●
	orp_{μ}	●	●	●	●	●	●	●
	$depth$	●	●	●	●	●	●	●
entropy= moderately high	in			●	●	●	●	●
	orp_{α}			●	●	●	●	●
	T			●	●	●	●	●
	V			●	●	●	●	●
entropy= low	$andp_{\mu}$					●	●	●
	$isTree?$					●	●	●
	no_{α}					●	●	●
	no_{μ}					●	●	●
entropy= none	$andp_{\alpha}$							

Fig. 6. Circles denote that a variable appeared in a learnt decision tree. Variables with low entropy (information content) disappeared from the learnt trees as m increased.

$andp_{\alpha}$ never appeared in the learnt trees. That is, to a accuracy level of (100-11.5=88.5%), skews in the number of and-node parents do not effect our ability to reach some portion of a system.

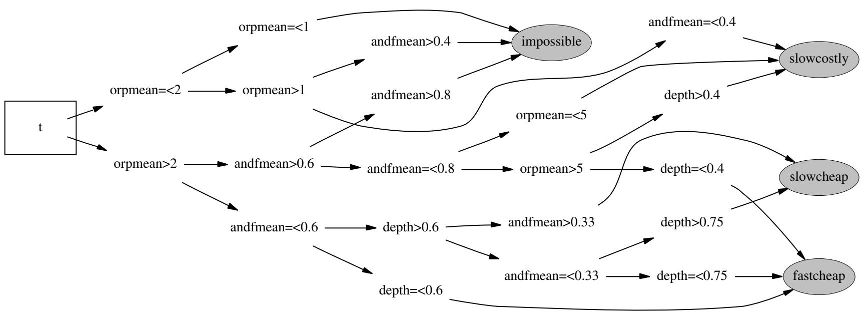


Fig. 7. A decision tree at $m = 4096$: 21 nodes, estimated error=32.3.

4 Discussion

In $(36 + 20 = 56\%)$ of our simulation runs, a small number of randomly selected inputs will reach as many parts of the graph as a very, very large number of

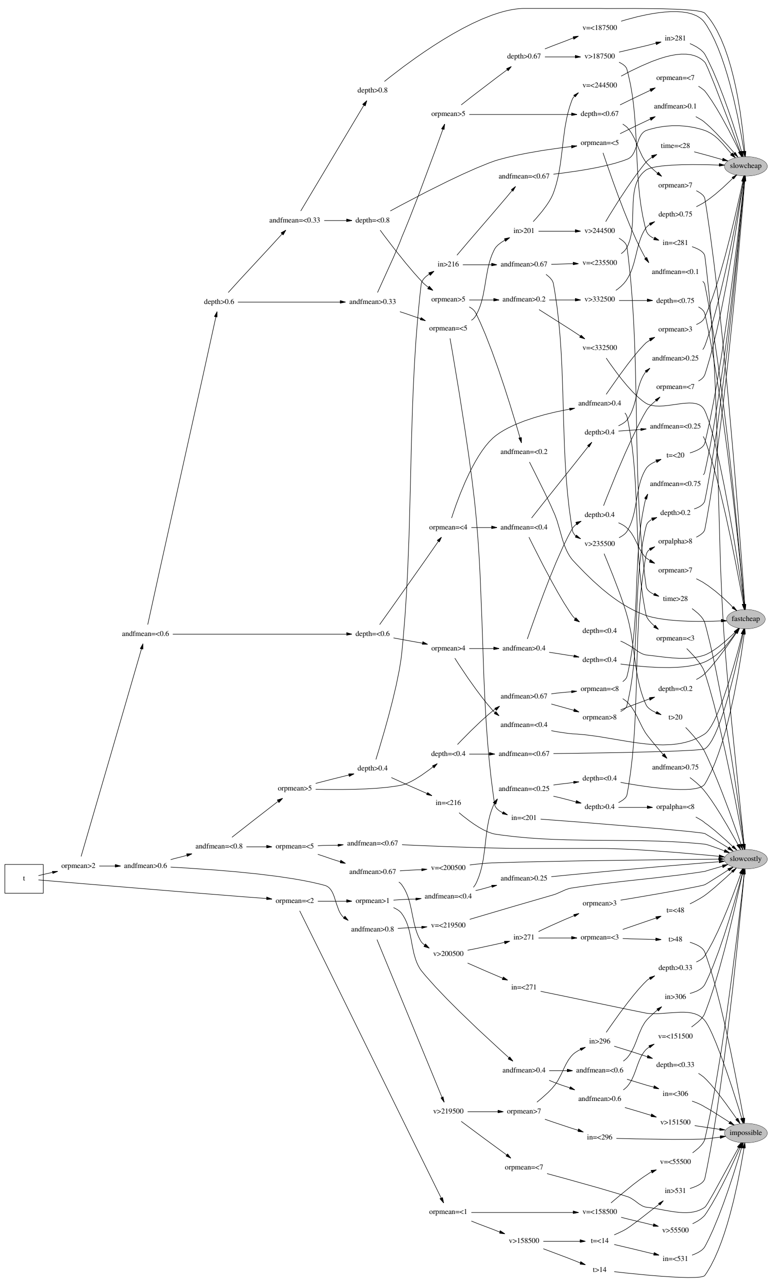


Fig. 8. A decision tree at $m = 512$: 117 nodes, estimated error=23.3.

inputs. While this result seems counter-intuitive, analogous results have been frequently reported in the literature. Randomized search can quickly cover as much of a knowledge-based system as a slower rigorous search [16]. Tests rapidly *saturate* a procedural program; i.e. after a small number of tests, the faults found by further testing rapidly decrease. Further, even after extensive testing, it is often found that large portions of a procedural program were not reached by the test suite [9]. Such a result is consistent with programs containing components that are either *fast and cheap* to reach or *impossible* to reach. For a full survey of these analogous results, see [11].

The high information value of *depth*, and the moderately high information value of *in* is disappointing. Parameters such as $andf_\mu$ and orp_μ can be extracted from a static analysis of the source code (for a discussion of difficulties in this extraction process, see [14]). However, *depth* and *in* can only be determined when a system executes over some inputs. Hence, static analysis to determine testability will only ever be an approximation.

The low information content of the *no* parameters is very surprising. Figure 6 shows that we can generate approximations of our model by ignoring *no* and that such approximations are within 6% of the maximal accuracy. In safety-critical domains, this 6% lose in accuracy would be unacceptable. However, for non-safety-critical domains, there seems little to lose in ignoring the distinction between determinate and indeterminate systems.

5 Conclusion

Agents can talk faster and consume less resources if they are in consensus. As agents work with each other, they should continually re-test for consensus. The bandwidth between agents in consensus can be minimized, but only if the continual re-testing process can be performed with minimal exchanges.

We have argued that declarative and procedural agents can be modeled as NAYO graphs (recall Figure 2 and Figure 4). Next, we characterized testing as generating pathways over NAYO graphs to reach error conditions. A mathematical model of average-case NAYO graph reachability was then presented and simulated 100,000 times. In just over half of those simulations, we found that testing is either *fast and cheap* or *impossible.* That is, half the time, what we learn very quickly about other agents is all we can hope to learn for a very long time. However, around a quarter of the time, it will be *slow and expensive* to test for consensus in another agent. Hence, if you work with these *slow and expensive* agents for a very long time, then a non-consensus attitude may suddenly emerge.

A clear research direction from this work is the further definition of language features that simplify the time required to check consensus amongst agents. Recall that certain runtime parameters (*depth* and *in*) determine some of the testability. Hence, these language features will only ever be an approximate guarantee of increased testability. Nevertheless, our experimentation continues. For example, Figure 8 suggests that the more or-node parents, the easier it will be to test for consensus. A partial list follows of language features that generate NAYO graphs with high orp_μ is:

– Disjunctions (obviously).
– Conditionals that use a randomly generated number and a threshold comparison.
– Polymorphism: one “or” would be generated for each type in the system that is accessed by this polymorphic operator.
– Calls to methods implemented and over-ridden many times in a hierarchy: one “or” would be generated for each possible received of the message.

Acknowledgements. This work was partially supported by NASA through cooperative agreement #NCC 2-979.

References

1. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
2. E. Coiera. Communication in Organisations. Technical report, Hewlett-Packard Laboratories, 1996. Technical Report HPL-96-65, May, 1996.
3. E. Coiera. Communication under scarcity of resources. Technical report, Hewlett-Packard Laboratories, 1999. Technical Report, 1999, to appear.
4. E. Coiera. When Conversation is better than computation. *Journal American Medical Informatics Association*, 7:277–286, 2000.
5. H.N. Gabow, S.N. Maheshwari, and L. Osterweil. On Two Problems in the Generation of Program Test Paths. *IEEE Trans. Software Engrg*, SE-2:227–231, 1976.
6. D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.
7. S. Brennan H.H. Clarke. Grounding in Communication. In *Perspectives on Socially Shared Cognition.* American Psychological Association, 1991.
8. G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
9. J. Horgan and A. Mathur. Software Testing and Reliability. In M. R. Lyu, editor, *The Handbook of Software Reliability Engineering*, pages 531–565, McGraw-Hill, 1996.
10. J. C. McCarthy and A. F. Monk. Channels, conversation, co-operation and relevance: all you wanted to know about communication but were afraid to ask. *Collaborative Computing*, 1:35–60, 1994.
11. T. Menzies and B. Cukic. On the Sufficiency of Limited Testing for Knowledge Based Systems. In *The Eleventh IEEE International Conference on Tools with Artificial Intelligence. November 9-11, 1999. Chicago IL USA.*, 1999. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-011.pdf`.
12. T. Menzies, B. Cukic, and E. Coiera. Smaller, Faster Dialogues via Conversational Probing. In *AAAI'99 workshop on Conflicts and Identifying Opportunities.* Available from `http://research.ivv.nasa.gov/docs/techreports/`, 1999.

13. Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Indeterminate Systems, 2000. Submitted ISSRE 2000.
14. G.C. Murphy, D. Notkin, and E.S.C. Lan. An Empirical Study of Static Call Graph Extractors. Technical Report TR95-8-01, Department of Computer Science & Engineering, University of Washington, 1995.
15. J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.
16. B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI '92*, pages 440–446, 1992.

Refining Interactions in a Distributed System

Neelam Soundarajan

Computer and Information Science
The Ohio State University
Columbus, OH 43210, USA
neelam@cis.ohio-state.edu

Abstract. We identify a type of refinement that we call *interaction refinement* that is very useful in designing distributed systems. An interaction refinement step is one in which a system designer decides to implement a given 'high-level' interaction as a particular sequence of 'low-level' interactions. For example a particular high-level interaction may be specified as '*identify-client*' involving a 'client process' and a 'server process'. The designer may decide to implement this interaction as a series of low-level interactions between the two processes, with the client process first sending to the server process an *id*, followed by a *password*, and perhaps allow the client several attempts at getting the password right, etc. Here the designer is not refining the *internal* structure of either the client process or the server process; rather he[1] is implementing the 'identify-client' interaction between the two processes in terms of a sequence of 'low-level' communications between them. By contrast, other types of refinement that are normally considered are concerned with refining the internal structure of one or more of the processes, the interactions between the processes being taken as a given.

We motivate the idea of interaction refinement with simple and natural examples and develop a preliminary formalism that can be used to establish correctness of systems developed using interaction refinement. We also compare our work with previous work which allow one to deal with restricted kinds of interaction refinement.

Keywords: Interaction refinement; Communication traces; Correctness criteria.

1 Introduction and Motivation

Two types of refinement that have been studied extensively in the literature are what we will call *procedural refinement* (*PR*) and *structural refinement* (*SR*). In a step of *PR* we refine the task of achieving a given goal starting from a specified initial condition, into subtasks for achieving subgoals, such that the given initial condition implies the initial condition of the first subtask, the subgoal achieved by the first subtask implies the initial condition required by the second subtask,

[1] Following standard practice, we use 'he', 'his', etc. as abbreviations for 'he or she', 'his or her' etc.

etc., and the subgoal of the last subtask implies the original required goal. The process of refinement is repeated with each of the individual subtasks until the final subtasks can all be directly implemented using the primitives of the given programming language. *PR* is a powerful tool and is used extensively, especially in designing sequential programs.[2] Formal methods developed by Hoare, Dijkstra, Gries, and others, allow us to reason precisely about such programs, and the reasoning closely models the refinement steps used in developing the program.

While *PR* is concerned with refining the procedure for achieving the given task, *structural refinement* (*SR*) allows us to refine the *structure* of the system being designed. Thus, for instance, when designing a process P that will exhibit a certain communication behavior with external agents, we may decide to structure P into a pair of communicating processes P_1 and P_2 that will communicate with each other and with the external agents in such a way that when the internal communications are abstracted away, the resulting communication behavior that we see is exactly what was specified for P. Formal methods such as those of [10, 1,9,11] formalize this notion of structural refinement.[3]

The goal of this paper is to explore another type of refinement which we call *interaction refinement* (*IR*). To see what we mean by *IR*, let us consider an example: Suppose we are designing a system that will include two processes A and B and that the process A is expected to send a sequence of files to the process B. For instance, these processes might be part of a network of workstations, the process A being say a text editor on one of the workstations in the network, and B something like a file server that will save the files that A sends to it. A designer with experience in such systems might decide to have A send *differential* information, i.e., information on how each file differs from the previous one it sent, rather than complete files. In those cases where A is indeed something like an editor, this would result in an enormous reduction in network traffic; and since B could easily combine this differential information with the file it currently has to obtain the new file, it serves essentially the same purpose as the original specification. But according to most current formalisms, this would not be a correct refinement since the refined communications between A and B are not complete files as was originally required; yet most designers would agree that this is a correct, and even a good refinement. This paper presents an approach

[2] Procedural refinement is often also called *stepwise refinement*; we prefer the term procedural refinement to distinguish it from the other types of refinement, which are also stepwise, that we consider in this paper.

[3] Another type of structural refinement is used in Object Oriented systems which may or may not involve any concurrency; here we refine the structure of an object to include internal components that interact with each other in such a way that the overall behavior of the object is appropriate. Although the interactions between the internal components are not communications in the sense that the term is used in distributed systems, the OO literature because of the similarity to communications between processes, often uses terms like 'messages' to characterize these interactions. Many of the ideas presented in this paper can also be applied in the OO setting [12]. Indeed in this paper we often use the terms 'object' and 'process' interchangeably.

to formalizing this type of refinement and for validating the correctness of such refinements.

Consider another example: Suppose we were designing a system consisting of a 'client object' CO and a 'server object' SO. The high-level specification of the system might say that the purpose of the first interaction between these two objects is to identify the client to the server. The system designer may map this into a series of interactions in which CO provides its 'user-id' and a 'password'; if CO provides an invalid password, SO might allow it several chances to get the password right; etc. The designer may also introduce a new object –an authentication object AO– to mediate this interaction. The authentication of the password would be encapsulated inside AO. Indeed it might be a better design to have CO send the password directly to AO rather than via SO, since the protocol on how many attempts the client is allowed to get the password correct etc., can then be encapsulated in AO which is probably where it belongs. In any case, what we have done here is to refine the high-level interaction *identify-client* between CO and SO into a series of interactions involving CO, SO, and AO. Again this refinement would not be considered correct by most formalisms since the interactions between CO and SO have been modified.

There is an important difference between *IR* on the one hand, and *PR* and *SR* on the other. In both *PR* and *SR* what we refine is *internal* to the object or process being designed; the *external* behavior is not being modified, indeed that is almost the definition of the correctness of a step of *PR* or *SR*. In *IR*, as we saw in the examples, this is no longer true. What we refined in the first example was not internal to the process A or to the process B; rather we modified the externally visible interactions between these two processes. The reason such a modification seems correct is that it seems to be consistent with the *intent* behind the original specification. Similarly in the second example, we did not talk about the internal structure of either CO or SO but rather about how to change the interactions between these objects, and even introducing a new object AO and interactions between CO and SO and this new object, in order to achieve the intent behind the original specification. Intentions are, of course, hard to formalize and we do not claim to do so in this paper. Rather, we believe that our formalization of *IR* will allow us to justify the types of refinements contained in these examples.

We will use a *trace* based approach in our formalization.[4] For simplicity we will also assume that a process is characterized by an invariant on its trace. The question that we are interested in addressing may then be stated as follows: If S' is obtained by an interaction refinement of S, what relation must hold between the traces associated with the processes of S' and those of S, and what relation must hold between the invariants of S' and S in order for us to consider this step of interaction refinement to be correct?

A number of authors have considered ideas that seem to be related to our notion of interaction refinement. Thus, for instance, Brinskma et al [2] consider

[4] We could use other models such as *failures* that would provide more information than traces can; however our goal in this paper is to explore the idea of interaction refinement in a relatively simple setting, so we prefer to use just traces.

what they call *interface refinement* as a method of refining the interface, i.e., the alphabet of interactions, between two processes. Our notion of interaction refinement seems more general and has to do with refining the entire sequence of interactions that a process or a group of processes go through; indeed, as we saw in the second example, we are even allowed to introduce new objects (or processes) to mediate the interactions between the original objects. Gerth et al [7] also use the term 'interface refinement' but they seem to be interested in a somewhat different problem. They want to be able to answer questions such as 'under what conditions is it legal to replace synchronous communications with asynchronous ones?', and we do not consider such refinements in this paper. But both [2] and [7] have one important thing in common with our work: the refinements they consider also manifest themselves in the *external* behavior of the object(s) in question, rather than being refinements of the internals. We will return to the relation of our work to that of others later in the paper.

The main contributions of the paper are the following:

– Present the idea of interaction refinement, and argue that it is useful in the design of distributed systems.
– Present a (preliminary) formalization of interaction refinement in the form of a simple proof rule, that allows us to verify the correctness of a particular refinement.
– Apply the formalism to a simple example to show its usefulness; the example we consider is a standard communication protocol, the Alternating Bit Protocol.

The rest of the paper is organized as follows: In the next section we present our approach to specifying processes, a fairly standard one using invariants over traces, and then proceed to define precisely what we mean by interaction refinement. We propose a rule for establishing the correctness of a step of *IR*. In section 3 we apply our approach to a series of examples, a standard set of communication protocols; each protocol is shown to be obtained by an *IR* step from the previous one. In the final section we summarize our work, compare it with other related work, and point out open problems that remain to be investigated.

2 Specifying and Verifying Interaction Refinements

We will use a *trace-based* approach to specify processes. With each process P we will associate two trace variables: ε will record all external interactions that P engages in; δ will record both external and internal interactions of P. If P has no internal processes, these two traces will of course be identical to each other. In general, ε can be obtained by projecting out of δ all elements that correspond to internal interactions of P. We will occasionally use τ to stand for either kind of trace, and let the context indicate whether we are considering the internal or external trace. (If we are considering the trace of more than one process, we will subscript the τ with the identity of the process to distinguish between them.)

An *interaction* is often, but not always, a *communication* in which one process sends a specific value to another. In general, low level interactions tend to be communications, while higher level interactions tend not to be. Thus in the example of the client process *cp* and server process *sp* of the last section, 'identify-client' is a high level interaction but not a communication, whereas the low level interaction in which *cp* sends a password to *sp* (or to *ap*) is a communication.[5] We will use an element of the form (c, v) in a trace to record a communication in which the value v was sent (by a process P) on the channel c.[6] We will record an interaction that is not a communication as an element that specifies the name of the interaction, the identities of the processes involved, and any parameter values that may be needed. Note that a communication involves exactly two processes, whereas a general interaction may involve two or more processes.

The specification of a system of processes will be an *invariant* on the traces of the processes.[7] Consider a system S consisting of a number of processes. A high-level specification of this system would be an invariant I on δ imposing appropriate restrictions on what interactions may take place between the component processes of S, as well as the interactions between S and external processes, and the relative orders of these interactions. Suppose we obtain S' from S following a step of interaction refinement, and suppose we have a specification I' for S'. The question from section 1 then is: What relation must hold between the traces of S and S' and between the specifications I and I' in order for us to be able to legitimately claim this step of interaction refinement was correct?

Note first that the obvious answer, that I' is a refinement of I only if it implies I, is wrong or rather not general enough. The main problem is that it does not account for the fact that the traces that these assertions impose conditions on do not contain the same kinds of elements. Rather the former correspond to what we have called high-level interactions, whereas the latter are low-level interactions (possibly communications) that *implement* those high-level interactions. So even if I' were a proper refinement of I, an implication relation will in general not hold.[8] In view of this difference between the elements in the traces, let us use δ to denote the trace that I refers to and δ' to refer to the trace that I' refers to. Note again that both δ and δ' are (internal) traces of

[5] This may be debatable; it may be that sending a password is actually quite a complex interaction involving a number of communications between the client and the server. This just goes to show that interaction refinement, like other types of refinement, is in general a multi-step activity.

[6] If communication is asynchronous, as in the example in the next section, we will treat the send and receive as two separate events, and record each in the communication trace.

[7] For simplicity we are considering only *safety* properties, thus invariants are sufficient. The main ideas in this paper will however apply to other kinds of properties than safety, and to other kinds of specifications than invariants.

[8] Of course, if I' does imply I, then I' is indeed a proper refinement of I but this is the simple kind of refinement that has been considered so far in the literature; we are trying to generalize this to the notion of interaction refinement.

S and record all interactions between the various component processes of S as well as interactions with processes external to S. The difference is that the view recorded in δ is a high-level view whereas that in δ' is a low(er)-level view.

Let us first consider a simple kind of relation between the elements in δ and δ'. Let Σ be the set of elements corresponding to the high-level interactions, i.e., that can appear in δ, and Σ' the set of elements corresponding to the low-level interactions. A simple situation would be for the interaction represented by any given element of Σ to be implemented by a sequence of elements of Σ'. Thus we will need a mapping of the form:

$$\rho : \Sigma \Rightarrow \Sigma'^*$$

where Σ'^* is the set of all (finite) sequences over Σ'.

If $\rho(i) = \langle i_1, \ldots, i_n \rangle$, that means that the high level interaction i is being implemented as the sequence of low-level interactions $i_1, \ldots, i_n$. Note that there is no assumption that only the processes involved in the interaction i are allowed to be participants in interactions $i_1, \ldots, i_n$. Indeed, it is not even required that all of the participants in i be involved in one or more of $i_1, \ldots, i_n$. This may seem rather strange; while it may be appropriate to allow additional processes to participate in the implementation of an interaction between a given set of processes, how can an interaction that at an abstract level involves certain processes be 'implemented' at a lower level without the participation of those processes? The answer is that this is a matter of system design and the formalism should not disallow such possibilities. For instance, it may be that the designer decided that some of these communications at the high-level were unnecessary and can be left out completely; or perhaps the nature of the system is such that this interaction is always preceded by certain other interactions involving certain other processes and hence these other processes can consult with each other to implement the required interaction without any apparent involvement of certain of the principals.

Once the mapping function is given, we can see what relation must hold between I and I'. Essentially we need to ensure that if a given trace δ' satisfies I', then the corresponding δ trace will satisfy I:

$$\forall \delta'. I' \Rightarrow [\exists \delta.(\delta' = \rho(\delta) \wedge I)] \tag{1}$$

where $\rho(\delta)$ is the sequence obtained by applying ρ to each element of δ and then appending together all the resulting sequences. Thus it is the natural extension of ρ which is defined over elements that correspond to 'high-level interactions' to sequences of such elements.

Rule (1) is a natural generalization of the usual implication relation between a specification I and its refinement I' to take account of the fact that during interaction refinement high-level interactions are implemented in terms of lower level ones. In particular if there is no difference between the high-level and low-level interactions, the function ρ reduces to the identity map, and (1) reduces to the usual implication relation.

But the mapping function ρ is a bit too restrictive. It requires that *every time* a given high-level interaction is implemented, it must be implemented in exactly the same way as the previous time. This would prevent refinements that exploit past history in implementing particular interactions. Consider again the example of an editor process and a file server. The high-level specification required a sequence of files to be sent by the editor to the file server, and in the refinement the designer decided to have the editor send differential information. Clearly we want to be able to allow such refinements as legitimate but the formalism we have developed so far will not do so, since according to the ρ function, a given high-level interaction must be mapped to a fixed set of low-level interactions.

The solution is clear from the discussion above. The function ρ should not be a function over the *individual* high-level interactions but rather over the high-level traces:

$$\rho : \mathit{\Delta} \Rightarrow \mathit{\Delta}'$$

where $\mathit{\Delta}$ is the set of all possible high-level traces (essentially the set $\mathit{\Sigma}^*$), and $\mathit{\Delta}'$ the set of all possible low-level traces.

But even this is insufficient. If ρ is a function, that means that a given interaction sequence at the high-level must be mapped to a fixed low-level sequence. In practice, it is quite possible for a given system, on different occasions, to implement a given high-level sequence via different low-level sequences. Thus, a more reasonable approach would be to allow ρ to be a *relation* between $\mathit{\Delta}$ and $\mathit{\Delta}'$. This also requires us to change rule (1) appropriately:

$$\forall \delta'. I' \Rightarrow [\exists \delta. (\rho(\delta, \delta') \wedge I)] \tag{2}$$

Note that (2) does not require that *every* δ that is related to δ' must satisfy the high-level invariant I, if δ' satisfies I', only that there must exist at least one such δ.

This formalization should not be taken to be the final version. Thus, for instance, we could argue that (2) is really vacuous; because we could simply define ρ to be the empty relation! In that case, (2) doesn't require us to establish anything, so effectively any system would be a refinement of any other! Clearly, this is not what we want. Perhaps we could require that *every* trace δ' that satisfies I' must be related to at least one trace δ (which would then be required to satisfy I according to (2)). We will not do that. Instead we will see a somewhat better solution, towards the end of the next section, following an example which cannot be handled if we were to require that every trace that satisfies I' be related to at least one trace that satisfies I.

3 Examples

In this section we will apply our approach to a series of communication protocols, each obtained, as we will see, by interaction refinement from the previous one. Our goal is not to provide a complete formal proof of the correctness of these protocols, but rather to show how *IR* plays an important role in the design of such systems and how our approach can be used to justify the individual steps of interaction refinement. The protocols we consider are taken from chapter 4 of Holzmann's book [8] on the subject. Holzmann also presents each protocol as an *informal* refinement over the previous one. Our goal is of course to formalize the intuition behind such refinement steps.

In our discussion we will follow (part of) the sequence of refinements that Holzmann presents. He starts with a simple protocol, called the *No-Flow-Control* protocol, where the sender process S simply sends a sequence of messages, which we will call *data* messages to distinguish them from other messages to be introduced shortly, and the receiver process R accepts them. Since the processes Holzmann considers are asynchronous, this can obviously lead to buffer overflow. So the next refinement[9] he introduces is the *Ping-Pong* protocol. Here S sends a data message, and waits for an *acknowledgment* message from R before sending the next message. R, similarly, receives a data message, sends the acknowledgment message, and waits for the next message. Note the interaction refinement here: the original protocol only talked about data messages being sent by S to R; in our refinement, we have introduced *new* messages, in the form of acknowledgments sent by R to S.[10]

While the *Ping-Pong* protocol takes care of the problem of buffer overflow, it does not account for the possibility of messages being lost (or delayed) in the channels. The problem cannot be fixed by simply introducing timeouts and having S re-send the message if it doesn't receive an acknowledgment after a period of time because then there would be a problem of matching the acknowledgment and the message being acknowledged. This leads to the next refinement, the famous *Alternating Bit Protocol.* Here we refine the data messages to include a 1-bit sequence number, so that alternating data messages have the sequence numbers 0, 1, 0, ... Also, the acknowledgments carry the sequence number of the message being acknowledged. The sender S sends a data message, including the sequence number and waits for an acknowledgment with the same sequence number. If within a fixed period, no acknowledgment is received, or only acknowledgments with the 'wrong' sequence number are received, S re-sends the message (including the sequence number). If the expected acknowledgment is received, S can be sure that the data message has been received by the receiving process R, and can proceed to the send the next data message. R similarly waits for a data message with an appropriate sequence number, 0 or 1. If it receives a message with the incorrect sequence number (1 when 0 was expected, or vice-versa), it sends an acknowledgment with the sequence number it received, and continues to wait; this corresponds to the case when the previous acknowledgment it sent

[9] Holzmann's first refinement is actually the *X-on/X-off* protocol; we will skip this protocol.

[10] If these new messages had been *internal* to either S or R, this would be an instance of structural refinement, but since the messages are exchanged *between* the two processes we are considering, what we have here is an interaction refinement step. In the *X-on/X-off* protocol that we skipped, not only are new messages between S and R introduced, but also each of S and R is split into a pair of parallel processes; the former is an instance of IR, the latter an instance of SR.

was not received by S. If R receives a message with the expected sequence number, S sends an acknowledgment with this sequence number, proceeds to deal with the received message, and then waits for the next data message. Note that this (interaction) refinement step has resulted in two distinct changes; first, data messages have been modified to include acknowledgment numbers; second, the acknowledgment messages have been modified to include a number. [11]

We will write down an invariant for the *No-Flow-Control (NFC)* protocol and take that as our starting specification. Next we will write down an invariant for the *Ping-Pong (PP)* protocol and show, using the approach of section 2, that it is a proper refinement of the *NFC* invariant. Finally, we will write the invariant for *ABP* and show that it is a proper refinement of the *PP* invariant. Note that under the usual definitions of correctness, such refinements would not be allowed; the specification of *NFC* has no mention of acknowledgment messages, so standard approaches would not let us treat (the invariant corresponding to) a protocol like *PP* as a refinement of that specification. But from a designer's point of view, one could reasonably argue that *PP* is indeed a (interaction) refinement of *NFC* and that *ABP* is a refinement of *PP* and that is exactly the point of this paper.

The invariant for each of these is relatively simple. Consider *NFC*. All we need, and can, say is that the sender S sends out a sequence of values, and the receiver R receives some of these values. Let a be the output channel of S and b the input channel of R. The (internal) trace δ of $[S \parallel R]$ will be made up of elements of the form (a, v) representing a value v being sent out on channel a (by S) and of the form (b, v) representing the value v being received on channel b (by R). The invariant I_N of *NFC* should assert that the values received on b were sent out on a. We cannot state, however, that the sequence $\delta_{/b}$, the sequence obtained from δ by retaining only the values communicated on elements of the form (b, v), is a prefix of the sequence $\delta_{/b}$, since *NFC* is supposed to allow message losses. Let us introduce the notation $s \prec\!\prec t$ to mean that if some (possibly none) of the elements of t are omitted, then the resulting sequence is equal to s; in other words, that the elements of s are all from t and in the same order as in t, but if two elements of t appear in s, that does not require that all the intervening elements also do. The invariant for *NFC* is:

$$I_N \equiv [\delta^N_{/b} \prec\!\prec \delta^N_{/a}]$$

We have used δ^N to denote the trace of *NFC* since we will also be dealing with the traces of *PP* and *ABP*; $\delta^N_{/b}$ of course denotes the sequence obtained from δ^N by retaining only the values communicated on b.

Note that this invariant allows elements sent out on a to be lost, but not du-

[11] Holzmann considers some other refinements. One of these, the *sliding window protocol (SWP)*, allows S to send several messages while waiting for the acknowledgments of earlier messages. The *SWP* is interesting because, as we will see towards the end of this section, it suggests that the formalization of *IR* that we have proposed in section 2 is not general enough. We will briefly discuss a possible way of extending the formalism to cover such refinements.

plicated or modified. It would be possible to write an invariant that allows for these also, but we will not do so here.

Next consider *PP*. The elements in the trace of *PP* occur in a very precise order: first a value is sent (by S) on a, then it is received (by R) on b, then an acknowledgment is sent (by R) on a new channel, let us call it α, then the acknowledgment is received (by S) on another new channel, call it β; and then the whole pattern repeats. The invariant I_P will just assert this; thus it will consist of four clauses, one of which, for instance, will be of the form:

$$\forall k \leq |\delta^P|.[(k \,\texttt{mod}\, 4) = 2] \Rightarrow [chan(\delta[k]) = b \wedge val(\delta[k]) = val(\delta[k-1])]$$

where *chan* is a function which returns the identity of the channel corresponding to a given element of the trace; and *val* returns the value communicated in the element. And δ^P denotes the trace of *PP*.

This clause states that the appropriate elements of δ^P correspond to (R) receiving on the channel b the value just sent (by S) on the channel a. We will omit the remaining clauses of I_P since they are exactly similar to this one.

In order to show that *PP* is a legal refinement of *NFC*, we must first define a relation ρ between the traces of *NFC* and *PP*. This is easily done; in fact, here we can define ρ as a function from the traces of *PP* to those of *NFC*: $\rho(\delta^P)$ is obtained by omitting all acknowledgment elements. It is easy to verify that if δ^P satisfies *PP*'s invariant, then $\rho(\delta^P)$ does satisfy *NFC*'s invariant.

This discussion also shows a weakness of the formalism: we could have gone through a similar argument if *NFC* had done nothing at all, i.e., no communications on any of the channels! Or if it had elected to stop after going through a certain number of communications. One could argue that that is because we are working with invariants and safety properties; and that if we were to extend the formalism to liveness issues, this problem would go away. But even then we would have had to make an important assumption in the case of *PP*; it would be correct only if we assumed that no messages can be lost. If any of the messages were lost in one of the channels, *PP* would deadlock and that presumably was not one of the intended behaviors of the original system. This is a question that needs further work.

Let us now consider *ABP*. As in the case of *PP*, we have four channels, corresponding to sending and receiving the value, and sending and receiving the acknowledgment. But both the values as well as the acknowledgments have a one bit sequence number, alternating between 0 and 1. The invariant is somewhat complex since it should allow for data values as well as acknowledgments to arrive in a far more arbitrary order than in the case of *PP*. Essentially, it should assert that: values are sent on a (by S) with alternating bits as sequence numbers; that this element may be received more than once (on the channel b, by the process R), but neither the data nor the sequence number is corrupted; that when R sends an acknowledgment on α, it must follow the receipt of a data item on b, and that the sequence number in the acknowledgment is the same as that in the item received on b; when a new value is sent by R on a, the acknowledgment with the appropriate sequence number must have been received; etc. We will leave it to the interested reader to write down the precise form of the invariant to allow for these various possibilities.

Next we have to define the ρ between *ABP* and *PP*. This is reasonably straightforward. All we have to do, given a trace δ^A of *ABP*, is to project out the extra data value messages that are sent (by S) following receipt of an acknowledgment with the wrong sequence number, and the extra acknowledgments sent following the receipt of a data value message with the wrong sequence number, and then remove the information about the sequence numbers from each of the messages. If we had written down the invariant of *ABP* we would then be able to complete the proof of the fact that *ABP* is a legitimate refinement of *PP* by showing that the corresponding sequence would satisfy I_P.

Before concluding this section, let us briefly consider *SWP*, the sliding window protocol. The idea of this protocol is to allow the sender to be several steps ahead of the receiver; in other words, even if the acknowledgment of a given message has not yet been received, the sender can send further messages (which are within the 'window'). As acknowledgments for various messages are received, the receiver can check off those messages and 'slide the window forward'. This would, of course, require bigger sequence numbers than just a single bit; indeed we can consider the *ABP* as a sliding window protocol with a window of size 1. It would seem reasonable to consider *SWP* as a refinement of *ABP* but there is a problem: Consider a trace of *SWP* in which the sender has sent, say, five data values, and has only received acknowledgments for the first one (presumably the window size was four or more). We cannot identify a corresponding trace of *ABP* that would satisfy *ABP*'s invariant because a new data value cannot be sent in that protocol before the acknowledgment for the previous one has been received. But note that ultimately the acknowledgments of the previous messages must be received in *SWP* also. A possible solution would then seem to be to require that if δ^S is a trace that satisfies the invariant of *SWP*, then there must be another trace δ'^S of *SWP* that also satisfies that invariant for which there must be a corresponding δ'^A that is related to δ'^S by ρ such that δ'^A satisfies the invariant of *ABP*. In other words, in order to allow for such refinements, what we need to do is to rewrite (2) so that what we require is the following: if a given system S' is a refinement of another system S, and if δ' is a trace that satisfies the invariant of S', we must be able to find a trace that δ' can *extend into* such that this latter trace also satisfies the invariant of S' and is related to a legitimate trace of S.

4 Discussion

We have identified *interaction refinement* as an important tool in the design of distributed systems. And we have provided a preliminary formalism that can be used to establish the correctness of systems designed using *IR*. It is worth repeating here that when using *IR* what we are doing is to decide to implement a desired 'high-level interaction' (or a sequence of such interactions) by a particular sequence of 'low-level interactions'. We are *not* designing the internal *structure* of any process, as we would be if we were using one of the other refinement procedures. While we do not have to worry about the internals of the processes, the fact that we are changing or refining the interactions between the processes means that we have had to redefine the notion of what it means for a refinement to be *correct*. Although we believe that the basic ideas underlying our formalism will prove useful, the examples in the last section show that some of the details need further investigation. The trick is try to make the requirements as unrestrictive as possible so that all refinements that we would intuitively consider as legitimate do indeed meet our requirements, but without going so far as to allow refinements that we would not consider legitimate. In the case of *PR* and *SR* this was relatively straightforward, since we could just require that the external behavior of the system remain unchanged. But here we are changing the external behavior, and what we are trying to maintain is the intention behind the original specification.

A different approach would be to accept *IR* as a design tool but not try to formalize it. In other words, we could simply throw away the original specifications once the refined specifications were arrived at. Indeed that is what is currently done. The motivation behind our paper has been that there is a reason why a designer performs a particular refinement, and why he thinks it is faithful to the intention behind the original specification. We believe that it is important to record such decisions, and record also the justification behind the decision in the form of the ρ function or relation and the proof that the two specifications are related as required by our formalism. Other designers will then be able to study such justification and understand the intentions of the original designer. Of course, if their interpretation of the intention behind the original specification is different, they may not agree with this designer's refinement, but at least the reason for this disagreement will be clear. By not retaining the original specification, or by not justifying the new design, none of this would be possible.

Several authors have considered ideas similar to *IR*. We have already mentioned the work of [2] and [7]. Broy [3] considers many different types of refinement (including what we have called procedural and structural refinements). One of these is called interaction refinement; but Broy's notion seems closer to the interface refinement of [2] rather than the general notion we are considering. Thus, it is not clear to the current author, whether, using Broy's definition the *SWP*, for instance, would be a legitimate refinement of *ABP*, or even whether *ABP* would be a valid refinement of *PP*.

Interestingly, there have been some attempts to establish the correctness of systems that have clearly used interaction refinement without depending on a formalism that allows for this. For instance, Creveuil and Roman [5] tackle the problem of verifying the correctness of a message router. They go through a series of design steps to arrive at their final router, and several of these steps are interaction refinements; in particular these steps introduce additional communications in the system. Creveuil and Roman formally establish using the approach of [4], that their router meets its specifications. They are able to do this because their router's high-level specification has been carefully designed in such a way that it *allows* additional communications –such as the ones introduced in their design– since it doesn't forbid them explicitly! Thus formally one can deal with this specification as if we are talking about a system that includes these types of communications, but that the specification imposes no special conditions on them! Clearly such an approach will not work if a particular interaction refinement in any way altered any of the high-level interactions, as in some of the examples we have considered. Even if that does not happen, it seems clear that not every instance where new communications are used can be treated using [5]'s approach.

Before concluding, we should mention also the work of Francez and Forman [6] on *Interacting Processes*. They generalize the notion of communication between two processes to a multi-party interaction. Their language notation allows us to define a *team* which is a kind of template for a particular kind of multi-party interaction. Then whenever a set of processes wishes to engage in this type of interaction, they all 'enroll' in the particular team. Is such a team the same as an interaction refinement? In other words, could we consider the multi-party interaction as a high-level interaction, refined by the low level details specified in the corresponding team? Although there seems to be some relation, we believe a team is not an interaction refinement. Rather, a team is a particular implementation of the various communications needed to achieve the multi-party interaction. It is certainly a refinement, but it seems to be more an instance of concurrency refinement, rather than interaction refinement.

References

1. K. Apt, N. Francez, and W. deRoever. A proof system for csp. *ACM TOPLAS*, 2:359–385, 1980.
2. E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In *Proceedings of TAPSOFT*, pages 297–312. Springer-Verlag, LNCS 494, 1991.
3. M. Broy. (inter)action refinement: the easy way. http://www4.informatik.tu-muenchen.de/paper_db/papers/broy_easy_way.html.
4. K.M. Chandy and J. Misra. *Parallel program design*. Addison-Wesley, 1988.
5. C. Creveuil and G.C. Roman. Formal specification and design of a message router. *ACM TOSEM*, 3:271–308, 1994.
6. N. Francez and I. Forman. *Interacting processes*. Addison-Wesley, 1996.
7. R. Gerth, R. Kuiper, and J. Segers. Interface refinement in reactive systems. In *CONCUR '92*, LNCS 630, pages 145–167. Springer-Verlag, 1992.
8. G. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.
9. J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Eng.*, 7:417–426, 1981.
10. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.
11. N. Soundarajan. Axiomatic semantics of csp. *ACM TOPLAS*, 6:647–662, 1984.
12. N. Soundarajan. Interaction refinement in the design of oo systems. In N. Debnath and R. Lee, editors, *Proceedings of the 3rd International Conf. on Software Engineering and Applications*, pages 269–274. IASTED/ACTA Press, 1999.

Design and Mathematical Analysis of Agent-Based Systems

Kristina Lerman

Information Sciences Institute
University of Southern California
4676 Admiralty Way,
Marina del Rey, CA 90292, USA
`lerman@isi.edu`
`http://www.isi.edu/~ lerman`

Abstract. Agent-based systems that are composed of simple locally interacting agents but which demonstrate complex group behavior offer several advantages over traditional multi-agent systems. A well-designed complex agent-based systems is an efficient, robust, adaptive and stable system. It has very low communication and computational requirements, meaning that there are virtually no constraints on the system size. The simplicity of agent interactions also makes it amenable to quantitative mathematical analysis. In addition to offering predictive power, mathematical analysis enables the system designer to optimize system performance.

To date, there have been relatively few implementations of complex agent-based systems, mainly because of the difficulty of determining what simple agent strategies will lead to desirable collective behavior in a large system. We claim that there exists a set of primitive agent strategies, similar to the basis behaviors in behavior-based robotics, from which complex group behavior can be designed. Moreover, these simple primitive strategies naturally lend themselves to mathematical description, making a quantitative study of agent-based systems possible. We present a case study of coalition formation to show that two simple behaviors, *dispersion* and *aggregation*, can lead to coalition formation in a multi-agent system under some conditions. We use this system to illustrate the process by which a mathematical description of the agent-based system is created and analyzed, and discuss the insights the analysis provides for designing coalition forming agents.

1 Introduction

Recent years have witnessed an explosion of interest in the study of agent-based systems, *i.e.*, systems composed of many interacting autonomous, artificial intelligent agents. Such systems may be used for distributed control (*e.g.*, network routing [24,4]), distributed resource management (*e.g.*, load balancing [23,9]), optimization [11], and electronic marketplaces [3,14]. Distributed robotics is a field of embodied agent-based systems that addresses the problems of coordinated action [12,15,16,6] as well as learning [18] in a group of robots. Agent-based computing has been introduced as the next step in the development of computation, in which autonomous proactive components replace the mostly reactive objects programmers create today [10].

1.1 Individual *vs.* Emergent Complexity

The central problem in the design of a multi-agent system is how much intelligence to place in the system and at what level. Does a central authority create and direct group behavior, or does the global behavior emerge from interactions among many individual agents? How complex should each agent's behavior be? The vast majority of the work in this field has focused on making agents more knowledgeable and able. This has been achieved in several ways: by giving the deliberative agent a deeper knowledge base and ability to reason about data [21], giving it the ability to plan actions [26], negotiate with other agents [20], or change its strategies in response to actions of other agents [23,8]. At the opposite end of the spectrum lie agent-based systems that demonstrate complex group behavior, but whose individual elements are very simple. Such systems have not received much attention in the agents community.

It has long been recognized in physics and biology that complicated global activity can result from very simple local interactions. Examples of this phenomenon pervade the natural world and include among others: pattern formation in thermal convection [13], Turing patterns [27] in chemical reaction-diffusion systems [19], the transition by which single cell slime mold amoeba aggregate to form a functioning multicellular organism [22].

Collective behavior emerging from local microscopic interactions has also been observed in many species of social insects. Though individual insects are arguably very simple creatures, having very limited memory, knowledge or reasoning facilities, an insect society demonstrates many complex behaviors. Hives, ant trails and swarms are examples of such robust, adaptable, seemingly "organized" collective behavior that does not have any central control [2].

1.2 Design of Agent-Based Systems

Complex agent-based systems (CAS) composed of simple agents that demonstrate complex collective behavior offer several advantages over traditional multi-agent systems that rely on deliberative agents. Though some problems are best suited for knowledgeable and able agents, traditional multi-agent systems (MAS) that utilize them have significant shortcomings in at least one of the following areas: robustness, adaptability, stability and scalability. Complex agents may fail, and if a central controller is involved in directing actions of agents, it has to be able to recover in the event of agent failure. Systems in which agents change their strategies in response to actions by other agents can quickly adapt to environmental changes; however, this feature is usually achieved at the expense of global stability [8]. The high communication and computational cost required to coordinate agent behavior constrain the size of the traditional MAS to at most a few dozen agents. Yet another disadvantage is that the complexity of the agent's internal states and its interactions with other agents make these systems ill suited for rigorous quantitative analysis.

A well-designed CAS, on the other hand, is an efficient, robust, adaptive and stable agent-based system. It lacks central conrol, meaning that the system can recover quickly from mistakes, agent failure and environmental change. Because it has very low communication and computational requirements, there are virtually no constraints on system size. This simplicity makes CAS amenable to mathematical analysis. Despite their numerous advantages, there have been relatively few implementations of CAS outside of distributed robotics. The scarcity is partially explained by the difficulty of designing a CAS. The designer, in a sense, has to reverse-engineer the problem, *i.e.*, determine what microscopic interactions, or basis behaviors, are necessary to produce the desired collective behavior [17].

Matarić [15] introduced basis behaviors as the fundamental components for behavior-based control in robots. A small set of primitive behaviors — collision avoidance, following, dispersion, aggregation and homing — is sufficient to synthesize complex behavior, such as foraging and flocking, in a single robot or a group of robots. We claim that a similar set of primitive agent strategies can be formulated for software agents, and they will serve as basic components for synthesizing collective behavior. Consider, for example, coalition formation. Coalition formation is a desirable behavior in systems where a group of agents can accomplish a task more effectively than a single agent can. The tasks may be very different – from collective block pushing, to commuter ride sharing, to consumers forming buying clubs to purchase products in bulk in order to save money – yet the underlying mechanism is always the same. We will demonstrate that coalition-formation in a system of software agents can result from two primitive agent strategies: dispersion and aggregation. Dispersion allows the agents to explore the environment in which they are situated and to encounter other agents and coalitions. Once an agent encounters a coalition, it makes a decision about whether to join it (aggregate). Other collective behaviors, such as distributed control and optimization (*e.g.*, task allocation), distributed resource management (*e.g.*, load balancing), collaborative information gathering, and cooperative transport in robots, may require the introduction of other primitive strategies.

1.3 Mathematical Analysis of Agent-Based Systems

Decomposing a complex collective behavior into simple individual strategies allows us to create a mathematical model of the process and to analyze the system quantitatively. The analysis helps the agent designer determine what the important parameters of the problem are, and their effect on the global characteristics of the system. Analysis also helps the designer decide how the primitive strategies should be put together to optimize the system performance. We will focus on the *macroscopic* models that directly describe collective behavior. For coalition-formation, for example, the macroscopic model describes the number and size of coalitions, not the behavior of individual agents. While some researchers have studied the behavior of agent-based systems quantitatively, they have almost exclusively focused on using *microscopic* simulations, such as molecular dynamics [5] and cellular automata, to model interactions between agents. Though microscopic simulations are an important tool for understanding the connection between the microscopic (agent) and the macroscopic (collective) behaviors, they offer only an indirect, empirical approach to studying the collective effects. Macroscopic models, on the other hand, directly describe collective behavior. Their other advantages are that they are more computationally efficient, because they use many fewer variables than the microscopic models; they are generic, meaning that the same model (with different parameters) is applicable to different systems, and they have better predictive power. Of course, microscopic and macroscopic theories are related, and understanding the connection between the two, *e.g.*, through simulation or by deriving the latter from the former, is an important goal of any complex systems research.

2 Coalition Formation in Agent-Based Systems

We present a case study of coalition formation in a multi-agent system to illustrate the process by which a mathematical description of the agent-based system is created, analyzed and what insights this analysis can provide for the system's design. The model we obtain is expressed mathematically as a series of differential equations that describe how the number and distribution of coalitions change with time. In addition to the dynamic variables, in this case coalitions, the equations contain variable parameters that determine how various agent strategies contribute to the behavior of the system. Many of the important global characteristics of the collective behavior, such as the existence of the steady state, the time it takes for the system to reach it, and the overall benefit of cooperation, depend on the values of these parameters. Information about the relationship between these parameters and the agent strategy is valuable to the designer of the system, who may want to control the global properties of a large-scale system.

Coalition formation is a valuable collective behavior in many systems. Consider a system where each agent is given a task to obtain goods at the lowest price [14]. We assume that agents know all the vendors that supply the requested goods and the retail (base) price for the product. Because bulk orders reduce manufacturer's costs, the vendors pass some of the savings to consumers. Therefore, the agents can lower the price they pay for the products by forming coalitions to buy them in bulk. Each agent moves among vendor sites. If it encounters other agents at a site, it can join the coalition or form a new one with a single agent. Manufacturing and other types of constraints limit the bulk order to a maximum size; therefore, vendors will not accept orders greater than this maximum.

The coalition formation mechanism outlined above is quite general and applies to any system in which it is beneficial for rational agents to form groups.

Moreover, this mechanism is local and requires minimal communication between agents. Agents learn indirectly about the presence and size of the coalition at a particular site by querying the vendor for the current price of the product. The agent knows the relationship between the size of the coalition and the expected benefit of joining it. This mechanism can be decomposed into two elements: dispersion and aggregation. Because these can be programmed directly into the agent's behavior, we call these elements primitive strategies or basis behaviors. The benefit of choosing these primitive strategies is that they are easy to describe mathematically, in addition to being relatively easy to implement. We list the primitive strategies below, together with additional simplifying assumptions that will make the construction of the initial mathematical model easier.

- *Dispersion:* Agents encounter other agents and coalitions randomly.
- *Aggregation:* Each agent's strategy is determined entirely by local conditions — the size of the coalition present at a particular site.
- Agents are homogeneous, in a sense that each agent has the same goal and follows the same strategy.
- The agents are spatially uniformly distributed, apart from the non-uniformities inherent in the coalitions. Even if this is not true initially, dispersion will tend to make the system uniform.
- There is no net change in the number of agents in the system.
- Agents are "mobile", *i.e.*, free to choose vendors; coalitions are not.
- It is beneficial for agents to join a coalition; however, an agent cannot join a coalition already of maximum size.
- Agents are self-interested; therefore, given several alternatives, they will prefer and select ones most beneficial to their goals.
- An agent may leave a coalition and find a better one to join.

Though the assumptions above leave us with a severely idealized system, the model that we create using these assumptions still contains all the important ingredients of the coalition formation process and will correctly capture the behavior of the system. We can make the model more realistic by incrementally relaxing these assumptions and adding more realistic agent behaviors.

3 The Macroscopic Model

Using the axioms above we can construct a microscopic theory of the coalition formation process, which treats the individual agents as the fundamental units. This model would describe how agents make decisions to join coalitions. Alternatively, we can construct a macroscopic model that treats coalitions as fundamental units of the system. A macroscopic description offers several advantages, the most important is that such a model directly describes the global properties of the system we are interested in studying, namely the number and size of coalitions, and how these quantities change with time. Macroscopic theories tend to be more universal—the same mathematical description can be applied to other systems in which aggregation occurs. However, the macroscopic model are usually phenomenological, and in some cases it may be necessary to derive the parameters of the model, and the model itself, from microscopic theory. This is not as vital in our application, because there is a simple connection between the microscopic behavior of the agents and the parameters of the model

We now present the macroscopic model that describes the time evolution of coalitions. The dynamic variables of the problem are the quantities we are interested in studying, namely coalitions, and they are labeled by their size. Let $r_1(t)$ denote the number of unaffiliated agents in the system at time t, $r_2(t)$ the number of coalitions of size two, *etc.*; $r_n(t)$ the number of coalitions of size n at time t, up to a maximum coalition size m.

3.1 Global Utility Gain

The global utility gain measures the efficiency of the system. For the e-commerce application, the total utility gain is the price discount all agents receive by being members of coalitions and was shown to be [14]

$$G = N\Delta p \left(\sum_{n=1}^{m} \frac{n^2 r_n}{N} - 1 \right) .$$

where Δp measures how steeply the vendors decrease the price for each new member of coalition. We derived this expression from the discount $\Delta p(n-1)$ received by each member of the coalition of size n. Note that this form of the utility gain applies only to the e-commerce application. For other problem domains another expression for the utility gain function may be necessary.

3.2 Dynamic Equations

Initially (at $t = 0$) the system consists of N agents and no coalitions. We assume that there is no spatial dependence in the agent distribution, apart from coalition-based aggregation; therefore, the variables are functions of time only. A series of coupled ordinary differential equations describe how the number of coalitions of different size changes in time; therefore, the solutions of the equations yield the coalition distribution at any given time:

$$\frac{dr_1(t)}{dt} = -2D_1 {r_1}^2(t) - \sum_{n=2}^{m-1} D_n r_1(t) r_n(t) + 2B_2 r_2(t) + \sum_{n=3}^{m} B_n r_n(t) . \quad (1)$$

$$\frac{dr_n(t)}{dt} = r_1(t) \left(D_{n-1} r_{n-1}(t) - D_n r_n(t) \right) - B_n r_n(t) + B_{n+1} r_{n+1}(t) , \quad (2)$$

$$\frac{dr_m(t)}{dt} = D_{m-1} r_1(t) r_{m-1}(t) - B_m r_m(t) \quad (3)$$

Here $r_n(t)$ is the number of coalitions of size n at time t, and $\frac{dr_n}{dt}$ is the rate of change of this number. Parameter D_n, the attachment rate, controls the rate at which unaffiliated agents join coalitions of size n. This parameter includes contributions from two factors: the rate at which agents encounter n-mers ($\propto r_1 r_n$, where the proportionality factor determines how many vendor sites an agent visits in a given period of time), and the probability of joining the coalition of size n. B_n, the detachment rate, gives the rate at which agents leave coalitions of size n. The solutions are subject to the initial conditions: $r_1(t=0) = N$ and $r_n(t=0) = 0$ for all $n > 1$.

3.3 Results

Initial investigation of the model focused on the uniform attachment–uniform detachment case: $D_n = D$, $B_n = B$ for all n. The results have shown that mathematical analysis is not only feasible, but also yields non-obvious results. To simplify the analysis, we rewrite the equations in dimensionless form by making the following variable transformations: $\tilde{r}_n = r_n/N$ (density of coalitions of size n) $\tilde{t} = DNt$, $\tilde{B} = B/DN$ (dimensionless detachment rate). When the equations are written in dimensionless form, only a single variable — $\tilde{B}$, the dimensionless detachment rate — governs the behavior of solutions. The equations were integrated numerically using Mathematica for $m = 6$ and different values of $\tilde{B}$. In all cases solutions reach a steady state, in which the distribution of coalition densities no longer changes. Figure 1 shows how the steady state density of coalitions of each size changes. The leftmost unconnected set of points are for the no-detachment case $\tilde{B} = 0$. The steady state at this point consists mostly of coalitions of size two and three, a quickly decreasing number of larger size coalitions, and no unaffiliated agents. When $\tilde{B}$ is small, $\tilde{r}_1$ is also small and the largest coalitions dominate. The number of unaffiliated agents, $\tilde{r}_1$, increases with $\tilde{B}$. Coalitions start to "evaporate" quickly at $\tilde{B} \approx 1$; as a result, the number of larger coalitions drops precipitously. Note that there is a discontinuity at $\tilde{B} = 0$: the steady state solutions are qualitatively different for $\tilde{B} \to 0$ than at $\tilde{B} = 0$. Moreover, for $\tilde{B} \neq 0$, the steady state is an equilibrium state: even though agents are continuously joining and leaving coalitions, the overall distribution of coalitions does not change. For $\tilde{B} = 0$, the system gets trapped in an non-equilibrium state before it is able to form larger coalitions.

The global utility gain (per agent) in the steady state, calculated according to Eq. 1, is shown in the lower half of Figure 2. The utility gain is largest for small non-zero $\tilde{B}$. Its value for $\tilde{B} = 10^{-6}$ is $G/N = 4.87\Delta p$ — a substantial increase over the no-detachment case value of $G/N = 2.00\Delta p$. For large detachment rates there is virtually no utility gain, as the system is composed mainly of unaffiliated agents. The large increase in the utility gain for small $\tilde{B}$ comes at a price, namely the time required to reach the steady state, plotted in the top half of Figure 2. While it takes $\tilde{t} \approx 10$ for solutions to reach the final state for $\tilde{B} = 10$, it takes $\tilde{t} \approx 10^9$ for the solutions to equilibrate for $\tilde{B} = 10^{-6}$.

The no-detachment ($\tilde{B} = 0$) case is especially interesting because mathematical analysis shows that global system properties, specifically, the steady state coalition distribution and the global utility gain, are outside of the agent designer's control. We have already shown using dimensional analysis, that coalition distribution is independent of the attachment rate, D. We claim that for

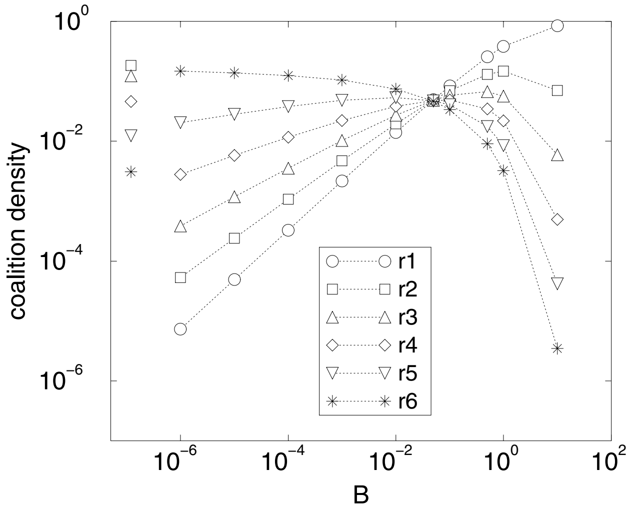


Fig. 1. Steady state distribution of coalition densities *vs.* the dimensionless detachment rate. The open symbols are coalition densities r_1 (unaffiliated agents) through r_6 (coalitions of 6 agents).

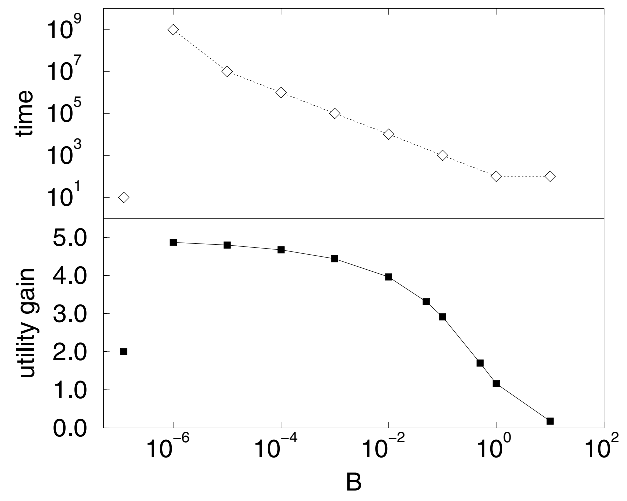


Fig. 2. The global utility gain per agent in the steady state vs. the dimensionless detachment rate and the time it takes for the system to reach the steady state.

the $\tilde{B} = 0$ case, it is also independent of m, the maximum allowed coalition size. We introduce a new variable — $\tilde{r} = \sum_{n=2}^{\infty} \tilde{r}_n$ — the total coalition density. As m becomes large, the first sum in Eq. 1 approaches $\tilde{r}$. Summing all equations for every coalition size and using the fact that the sum of derivatives is the derivative of the sum, allows us to rewrite the rate equations in terms of two variables only: $\tilde{r}_1$ and $\tilde{r}$.

$$\frac{d\tilde{r}_1}{d\tilde{t}} = -2\tilde{r}_1^2(t) - \tilde{r}_1(t)\tilde{r}(t)\,, \tag{4}$$

$$\frac{d\tilde{r}}{d\tilde{t}} = \tilde{r}_1(t)\tilde{r}_1(t)\,. \tag{5}$$

The first equation is very similar to Eq. 1, and while the second equation might seem counterintuitive at first glance, it has a very simple meaning: namely, the total number of coalitions changes only when two unaffiliated agents join to form a coalition of size two. Solutions of these equations behave exactly the same way as solutions for each coalition size. Figure 3(a) shows the time evolution of the solutions to the full equations with $B = 0$ and $m = 6$, while the dashed lines in Fig. 3(b) show the time evolution of solutions to Eqs. 4- 5. For comparison, we also plot the sum of coalitions of size two through six from Fig. 3(a). Although equations Eq. 4 and Eq. 5 are valid for large m, we can see that there is no appreciable difference in the steady state solutions, and therefore, the utility gain, already for $m = 6$.

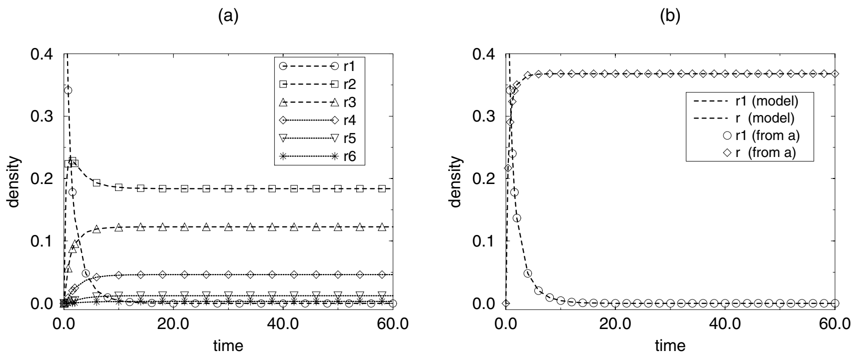


Fig. 3. (a) Coalition density *vs.* dimensionless time for the case where maximum coalition size is $m = 6$. (b) Solutions of equations for the single agent and total coalition densities *vs.* time (dashed lines). Single agent density (circles) and the sum of coalitions of size two through six (diamonds) in (a) are plotted for comparison.

We can try to repeat the analysis for the $\tilde{B} \neq 0$ case for large m by introducing a total coalition density variable and summing equations 1–3. Unfortunately, we can not eliminate every variable, and we are left with two equations and three unknowns: $\tilde{r}_1$, $\tilde{r}_2$ and $\tilde{r}$. This can be understood by considering that the total number of coalitions not only increases when two agents form a coalition of size two, but it can also decrease when a pair disintegrates. We can introduce a third equation for $\tilde{r}_2$. However, this equation involves a new variable, $\tilde{r}_3$, and to take *it* into account we need yet another equation, *etc.*, until we reproduce Eqs. 1–3. Fortunately, we can obtain analytic expressions for the steady state densities in terms of the monomer density from the rate equations by setting the left-hand side of Eqs. 1–3 to zero. We find that at late times the densities obey a simple relationship:

$$\tilde{r}_n = \tilde{B}^{-(n-1)}\tilde{r}_1^n \,. \tag{6}$$

By studying the behavior of solutions for different values of m, we empirically obtain a scaling law for the steady state monomer density,

$$\tilde{r}_1 \propto \tilde{B}^{\frac{m-1}{m}} \,. \tag{7}$$

This result is valid in the parameter range that we are interested in, namely where the utility gain is large and slowly varying. Equations Eq. 6 and Eq. 7, together, allow the agent designer to predict how the steady state density of coalitions of any size changes as the detachment rate or the maximum coalition size is changed. In particular, as maximum coalition size becomes large, the exponent of $\tilde{B}$ approaches 1. In this case $\tilde{r}_n \propto \tilde{B}$, that is the number of coalitions of every size grows linearly with $\tilde{B}$.

4 Lessons for Agent Designers

We have shown that, at least for the uniform attachment–detachment cases, a steady equilibrium state is reached for all non-zero values of the control parameter $\tilde{B}$. It is as yet unresolved whether the equilibrium state is stable. If individual agents are not allowed to leave coalitions (corresponding to the case $\tilde{B} = 0$), the steady state coalition distribution is independent of any of the system parameters that are under the designer's control. Even so, there is some utility gain in this system, reflecting the presence of small coalitions. Introducing even a very small detachment rate (or "shopping arount" rate) to the basic coalition formation process allows the system to increase the global utility gain by more than a factor of two over the $\tilde{B} = 0$ scenario. As the relative strength of the detachment rate increases, the utility decreases until there is virtually no utility gain. (This fact is known to every shopper who has spent the day at the mall running from store to store comparing prices and not ending up buying anything.) The price for higher utility gain is that the time required to reach the steady state solution grows very large as $\tilde{B}$ becomes small. However, utility gain remains large and decreases slowly over many orders of magnitude of $\tilde{B}$. The agent designer has much leeway in choosing parameter values that result in a substantial global benefit, while not requiring too long a wait for this benefit to be achieved. The agent designer can also predict the final distribution of coalitions, even for very large systems.

5 Related Research

Axelrod *et al.* [1] show how cooperative behavior can emerge among selfish autonomous agents. They use game dynamics to simulate interactions between two agents, in which the agents have to make decisions, each with a different pay-off or benefit to the agent. The agent's decision depends on choices made by other agents. Some strategies were shown to lead to stable cooperation, because it increases the overall pay-off to both cooperating agents. Others [23,7] have applied game dynamics formalism to distributed control, where many agents adjust their strategies (a decision to compete or to cooperate) to grab a larger share of a finite resource. The focus of this work is adaptation in a distributed system, *i.e.*, how a group of agents can learn to cooperate to achieve a common goal without any central control. Some of the systems (see, for example, Huberman and Hogg's work on computational ecologies [8,7]) are amenable to mathematical analysis, though most results about the stability of the system have been achieved through simulation. This line of inquiry is similar to ours, in a sense that it studies the global dynamics of a system of locally interacting agents. The aim of simulations, however, is to demonstrate the existence of evolutionary stable strategies that drive the system to the steady optimal solution, and how the agent's strategy evolves to maximize the benefit to itself. Our main goal, on the other hand, is understanding the global dynamics so that we can *control* the collective behavior of the system by manipulating individual agent's strategy.

Shehory *et al.* [25] have studied a large scale multi-agent system using a physics-based approach similar in spirit to ours. They suggest a low communication complexity coordination mechanism (though not coalition formation) for a large scale multi-agent system and use a physics-based microscopic model to analyze the system. Yet another physics-based approach used to study a multi-agent system is molecular dynamics simulations of swarm behavior [5]. In that work, the interactions among pairs of agents were modeled by a potential field. One advantage of microscopic simulations is that they allow agent behavior to be easily manipulated. However, simulations offer only an indirect way to study the collective behavior of the system, they are time consuming, and have little predictive power.

Acknowledgements. This research has been funded in part by the ISI-ISD Research Fund Award.

References

1. R. Axelrod and W. D. Hamilton. The evolution of cooperation. *Science*, 211:1390–1396, 1981.
2. Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence:* From Natural to Artificial Systems. Oxford University Press, New York, 1999.
3. A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology,* pages 75–90, 1996.

4. G. Di Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *J. of Artificial Intelligence Research,* 9:317–365, 1998.
5. R. Duncan, T. D. McCarson, R. Stewart, P. M. Alsing, R. P. Kale, and R. Ro bi nett. Statistical paradigms for robotic swarm modeling. http://www.mhpcc.edu/research/ab98/98ab41.html, 1998.
6. Dani Goldberg and Maja J. Matarić. Coordinating mobile robot group behavior using a model of interaction dynamics. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99),* pages 100–107, New York, May 1–5 1999. ACM Press.
7. O. Guenter, T. Hogg, and B. A. Huberman. Learning in multiagent control of smart matter. In *AAAI Workshop on Multiagent Learning,* 1997.
8. T. Hogg and B. A. Huberman. The behavior of computational ecologies. In B. A. Huberman, editor, *The Ecology of Computation,* Amsterdam, 1988. North-Holland.
9. Bernardo A. Huberman and Scott H. Clearwater. A multiagent system for controlling building environments. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems,* pages 171–176, San Francisco, CA, 1995. MIT Press.
10. N. R. Jennings and M. Wooldridge. Applications of Agent Technology. In N. R. Jennings and M. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets.* sv, March 1998.
11. J. Kennedy and R. Eberhart. Particle Swarm Optimizationy. In N. R. Jennings and M. Wooldridge, editors, *Fourth IEEE International Conference on Neural Networks.* IEEE Service Center, 1995.
12. H. Kitano, M. Asada, Y. Kuniyoshi, and I. Noda. RoboCup: A challenge problem for AI and robotics. *Lecture Notes in Computer Science,* 1395:1, 1998.
13. K. Lerman, D. S. Cannell, and G. Ahlers. Different convection dynamics in mixtures with the same separation ratio. *Physical Review,* E53:R2041, 1996.
14. Kristina Lerman and Onn Shehory. Coalition Formation for Large-Scale Electronic Markets. To appear in ICMAS'2000, 2000.
15. Maja J. Matarić. Designing emergent behaviors: From local interactions to collective intelligence. In J-A. Meyer, H. Roitblat, and S. Wilson, editors, *Second International Conference on Simulation of Adaptive Behavior (SAB-92)*, pages 432–441, 1992.
16. Maja J. Matarić. Designing and understanding adaptive group behavior. *Adaptive Behavior,* 4(1):50–81, December 1995.
17. Maja J. Matarić. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems,* 16(2–4):321–331, December 1995.
18. Maja J. Matarić. Learning in multi-robot systems. In Sandip Sen, editor, *IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems,* pages 32–37, 1995.
19. Z. Noszticzius, W. Horsthemke, W. D. McCormick, H. L. Swinney, and W. Y. Tam. Sustained chemical waves in an annular gel reactor: a chemical pinwheel. *Nature,* 329:??, 1987.
20. Simon Parsons and N. R. Jennings. Negotiation through argumentation – a preliminary report. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems.* MIT Press, 1995.
21. Simon Parsons, Carles Sierra, and Nick Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation,* 8(3):261–292, June 1998.
22. W.-J. Rappel, A. Nicol, A. Sarkissian, H. Levine, and W. F. Loomis. Self-organized vortex state in two-dimensional dictyostelium dynamics. *Physical Review Letters,* 83:1247–1250, 1999.

23. Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing: a study in co-learning. In Sandip Sen, editor, *IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems,* pages 78–83, 1995.
24. Ruud Schoonderwoerd, Owen Holland, and Janet Bruten. Ant-like agents for load balancing in telecommunications networks. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the 1st International Conference on Autonomous Agents,* pages 209–216, New York, February 5–8 1997. ACM Press.
25. Onn Shehory, Sarit Kraus, and O. Yadgar. Emergent cooperative goal satisfaction in large-scale automated-agent systems. *Artificial Intelligence,* 110(1), 1999.
26. Katia Sycara and Anandeep S. Pannu. The RETSINA multiagent system: Towards integrating planning, execution and information gathering. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98),* pages 350–351, New York, May 9–13 1998. ACM Press.
27. A. M. Turing. The chemical basis of morphogenesis. *Phil. Trans. Royal Soc. Lond.,* B327, 195.

Modeling Task and Teams through Game Theoretical Agents

Michel Rudnianski [1] and Hélène Bestougeff [2]

[1]University of Reims, ARESAD, 39 bis Avenue Paul Doumer ,
75116 Paris –France
Michel.Rudnianski@wanadoo.fr

[2]University Paris VII – Denis Diderot, University of Marne la Vallée
5 Boulevard Descartes- Champs sur Marne
77454 Marne La Vallée Cedex2 - France
bestoug@ladl.jussieu.fr

Abstract. Organizations are represented as conversational networks of agents, which interactions are modeled by a two-player game of deterrence, i.e. a qualitative game based on the concept of threshold. A congestion control algorithm is then derived from the game Boolean solution set. Last, analysis is extended to the case where the solution set is fuzzyfied. On the whole, the approach enables a mix-management : decentralized when agents can manage their relations successfully by themselves; centralized when the risk of congestion arises.

1 Introduction

There are a number of ways to design multi-agent based systems. We consider here agents as intelligent negotiating units which ultimate goal is to satisfy business objectives in a "reasonable fashion", while also satisfying their own goals and motivations. Therefore, we shall be looking for *satisficing* solutions, i.e. solutions which satisfy given constraints. In particular, we want to avoid congestion, that is a situation where the organization would be unable to carry out a single task.
An organization may be defined as a two-layer network : a conversational network and a task network. The first one defines the current and/or possible interactions between agents. The latter defines the set of constraints regarding the tasks. The system's dynamics is generated by carrying out selected actions, that can either trigger further interactions among agents, or activate tasks.
Agents behaviors are driven by their internal state and environment, as defined by the other agents and tasks sets. For instance, an agent requested to perform a task, can answer positively (commit) or turn down the request.
We shall represent interactions between agents at a given instant by a matrix, indicating the state of all bilateral relations, with respect to requests and commitments between agents.
In turn, we shall consider this matrix as the matrix of a two-player game of deterrence, the solutions of which should ensure that no congestion occurs.

This game theoretical framework will enable the transformation of a problem of interaction between, say N agents, into a two-player game.

More precisely, we shall first introduce the main properties of Games of Deterrence, a qualitative approach of interaction based on the threshold concept, and introduced by M. Rudnianski [7]. Games of Deterrence have been extensively studied, including N-player, dynamic [6] and fuzzy games [5]. Secondly, we shall show how a two-player game of deterrence can be used to analyze and simulate dynamic organizations. We shall then propose a control algorithm, the aim of which is to prevent congestion. Some preliminary results regarding the algorithm's efficiency have already been presented [4] . The present paper will extend these results, in particular, by fuzzyfying Games of Deterrence.

2 Games of Deterrence

2.1 Basic Definitions and Properties

We consider finite binary bimatrix games (S_R, S_C, A, B,S), with players Row (R), and Column (C), with respective strategic sets S_R (card S_R = n) and S_C (card S_C = p).
For any strategic pair $(r,c) \in S_R x S_C$, the outcome for Row is noted by $A(r,c) \in \{0,1\}$, and the outcome for Column by $B(r,c) \in \{0,1\}$.
Thus A(r,c) and B(r,c) can be interpreted as predicates, "1" indicating that the predicate is satisfied, i.e. the outcome of the player under consideration is acceptable, and "0" that it is not.
- A strategy r of Row is said to be *safe* iff : $\forall c \in S_C$, A(r,c). If r is not safe, r is said to be *dangerous*.
- A strategy r is said to be *positively playable* iff the predicate J(r) is true, where :

$$J(r) = (\exists r \in S_R)\, J(r) \wedge ((\forall c_i \in S_C)\, ((A(r,c_i) \vee (\neg J(c_i) \wedge (\exists c_j \in S_C)\, J(c_j)))$$

- A strategy $r \in S_R$ is said to be *playable by default* iff there is no positively playable strategy in S_R. In other words, the following predicate $\hat{J}$ (r) must be true:

$$\hat{J}(r) = \neg\ (\exists r \in S_R)\, J(r)$$

Similar definitions apply by analogy to strategies c of Column.

For any pair (s,s') of strategies of the same player, $\hat{J}(s) = \hat{J}(s')$. Therefore, from now on, $\hat{J}(s)$ will be denoted $\hat{J}_R$ if $s \in S_R$, and $\hat{J}_C$ if $s \in S_C$.
- The system S of all J(r), $r \in S_R$, and J(c), $c \in S_C$, is called the *playability system of the game*.
- A *solution* of S is a consistent set $\{J(r_1), J(r_2),..., J(r_n), J(c_1), J(c_2)...., J(c_p)\}$.
In the general case, there is no uniqueness of the solution.
- A strategy $s \in S = S_R \cup S_C$ is *playable* iff it is *either positively playable or playable by default.*
Given a strategic pair $(r,c) \in S_R x S_C$, r is termed *deterrent strategy vis-à-vis c* iff:

(1) $J(r) \vee \hat{J}_r$;

(2) $\neg A(r,c)$;

(3) $\exists c_i \in S_C$, such that $J(c_i)$.

It has been shown [7] that a strategy $c \in S_C$ is playable iff there is no strategy $r \in S_R$ deterrent vis-à-vis c. Thus, the study of deterrence properties amounts to analyzing the strategies playability properties.
- A binary bimatrix game (S_R, S_C,A,B,*S*) is termed a *matrix game of deterrence*, and a strategic pair $(r,c) \in S_R \times S_C$ is said to be an *equilibrium* of this game if both strategies are playable for some solution of the playability system.

2.2 Graphs of Deterrence

For large strategic sets, resolution of the playability system may prove lengthy. Therefore, we shall establish a correspondence between the playability system and a graph space in which the problem can be treated more simply.
Given a game of deterrence (S_R, S_C, A, B,*S*), we shall call *graph of deterrence*, a bipartite graph G on $S_R x S_C$ such that, given $(r,c) \in S_R \times S_C$, there is an arc of origin r (resp. c) and extremity c (resp. r), iff $\neg$ B(r,c) (resp. $\neg$ B(r,c)).
It has been shown that there is a one-to-one mapping between playability systems and graphs of deterrence (ibid).
Moreover, if we note $\Gamma^{-1}(s)$ the set of predecessors of a vertex s on G, and Δ (s) the logical function, true iff $\Gamma^{-1}(s)$ is not void, we get (ibid) :

$$J(s) = \neg\Delta(s) \ \vee (\neg \hat{J}(s) \wedge \forall s \in \Gamma^{-1}(s), [\neg[J(s') \vee \hat{J}(s')]])$$

Solving the playability system *S* then amounts to determining playabilities of the graph vertices. Since a graph can be decomposed into paths and circuits, we shall call :
- *R-path* (resp. *C-path*) a path the root of which is an element of S_R (resp. of S_C);
- *Primary circuit*, a circuit such that none of its vertices has an ancestor that does not belong itself to a circuit;
-*β-graph*, a graph, that includes neither an R-path nor an C-path.
It is shown (ibid) that :
(1) if G is an R-path, the only positively playable strategy for R is the root, while all strategies of C are playable by default;
(2) if G is a primary circuit, all strategies of both players are playable by default;
(3) if G is a β-*graph*, a solution of *S* satisfies :
(i) for any strategy s_0,

$$J(s_0) = \neg[\ \hat{J}_R \vee \hat{J}_E] \wedge \forall\ s \in N(s_0): J(s) \wedge \forall s' \in N'(s_0): [\neg J(s')]$$

where $N(s_0)$ (resp. $N'(s_0)$) is the set of the first strategies met when following G backward from s_0, and belonging to the same strategic set as s_0 (resp. to the other);
(ii) on a path, the vertices positive playability is determined by the parity of their distance to the first strategy met when following G backwards ;
(iii) each player has at least one non positively playable strategy.
Moreover, it is shown (ibid) that through appropriate cuts, it is always possible to decompose the graph of deterrence into connected parts, each one being an R-path, an C-path, or a β-graph. Hence, depending on the presence of these elementary components in the graph, one can distinguish 7 types of games : type R, type C, type β, type R-C, type R-β, type C-β, type R-C-β.

This typology leads in turn to the Classification Theorem (ibid) :

(1) Given a game of deterrence, its playability system's solution set is not empty.

(2) The game type defines the solution set.

It follows from (1) that every game of deterrence has an equilibrium, but the above shows that this equilibrium may not be unique.

With a small number of additional concepts, all previous definitions and results are extended to the case of N-player games[6].

2.3 Example 1

Rose and Charlie work for the same company, and may interact in order to fulfill their tasks or missions. First, each one can send the other a *request*, that is a demand for some help. Second, receiving a request from the other, each one can decide to answer positively or not. In the first case, we shall say that the agent *commits*.

Requests and commitments generate four possible attitudes, which may be considered as strategies in a game of deterrence :

- requests and commits ($R \wedge C$)
- requests and does not commit ($R \wedge \neg C$)
- does not request and commits ($\neg R \wedge C$)
- does not request and does not commit ($\neg R \wedge \neg C$)

The way each agent considers any pair of these strategies, that is, acceptable or unacceptable, defines the game matrix.

Let us suppose for instance that :

- Rose is a selfish person who hates to be bothered by a request for help, unless she needs some help herself, and therefore is willing to trade her help for Charlie's;
- Charlie is a man of essentially cooperative nature, who thinks that things can only get better, if everybody helps everybody.

The game matrix could then be given by Table 1.

Table 1. Example 1 Game matrix

Charlie

Rose		$c_1 = R \wedge C$	$c_2 = R \wedge \neg C$	$c_3 = \neg R \wedge C$	$c_4 = \neg R \wedge \neg C$
	$r_1 = R \wedge C$	(1,1)	(0,0)	(1,1)	(0,0)
	$r_2 = R \wedge \neg C$	(1,0)	(1,1)	(1,1)	(0,0)
	$r_3 = \neg R \wedge C$	(0,1)	(0,0)	(1,1)	(1,1)
	$r_4 = \neg R \wedge \neg C$	(1,0)	(1,0)	(1,0)	(1,1)

Strategy r_4 is safe, which implies that c_1, c_2, c_3 are not positively playable.

Moreover, analysis of the matrix shows that :

$J(c_4) = (1\text{-}J(r_1))(1\text{-}J(r_2))(1\text{-}\ \hat{\jmath}_c)$;
$J(r_1) = J(r_2) = (1\text{-}J(c_4))(1\text{-}\ \hat{\jmath}_c)$;
$J(r_3) = (1\text{-}\ \hat{\jmath}_c)$.
It follows that the game has two solutions:
(1){$J(r_1) = 0$; $J(r_2) = 0$; $J(r_3) = 0$; $J(r_4) = 1$; $J(c_1) = 0$; $J(c_2) = 0$; $J(c_3) = 0$; $J(c_4) = 0$}.
(2) {$J(r_1) = 0$; $J(r_2) = 0$; $J(r_3) = 1$; $J(r_4) = 1$; $J(c_1) = 0$; $J(c_2) = 0$; $J(c_3) = 0$; $J(c_4) = 1$};

If the players do not communicate about their strategic choices, Rose will always choose r_4, since it guarantees her an acceptable outcome, whatever the solution.
Charlie will not lose anything by playing c_4.
So, despite his cooperative mood, Charlie will adapt his behavior to the one of Rose : this is the well known conclusion according to which you need two to tango !

3 Communication Procedure between Agents

Communication procedure between agents is based on recurrent conversations and commitments. Conversations, as dyadic structured communication protocols between business roles, were introduced by Winograd and Flores. They have already been applied to workflow systems and business process reengineering [3].
Three features may be attached to an agent or a team : goal, action, motivation.

1. A *goal* is met through some predefined plan, that is a sequence of actions. Therefore, agents can reach different goals by using different plans and/or have several plans for a given goal.
2. An *action* can belong to two classes:
   - the class of tasks that the agent /team performs itself locally;
   - the class of goals that the agent transfers to another agent.

To be transferred to some other agent, a goal must be associated with a conversation. Each conversation is associated with only one goal, but a goal can be associated with different conversations (transferred in different contexts).
3. A *motivation* is an information which acts as a parameter for decision making.
Here, motivations determine acceptance or refusal of a goal by an agent.

Workloads are regularly sent to different teams. Each particular job is defined as a business goal associated with a conversation.
Conversations are meant to model not only communication, but also commitments : an agent will commit to a request if, either the latter meets its motivation, or commitment is made compulsory by the management, which exerts some hierarchical /social pressure.
Each agent has a limited work capacity, and may not be able to commit to more than few requests. Therefore, an important issue is congestion avoidance.

To analyze the dynamics of congestion, one needs to simulate the agent network, and assess the influence of the various parameters [1]. Time evolution is characterized by a sequence of events which triggers a change in the state of the system. Therefore, interaction between various agents is considered to occur at discrete instants of time, and handled by *events*. In the long run, whatever the agents flexibility, there must be some overall management control to prevent congestion.

4 Representation of Agent/Team Interactions through Games of Deterrence

At a given time t, each agent may send a request and/or a commitment throughout the conversational network. Hence, for any pair of agents (i,j), there are 4 possible *states* of i *with respect to j*, depending on whether i can send a request and / or a commitment toward j. Agent i can turn down a request already sent to j, who has accepted it, and so on. Thus, the model takes into account subtleties related to contradictions emerging between agents with time.

Requests and commitments serve a general purpose of the organization, namely satisfying demands of customers, located outside the conversational network. Thus requests may be originated either by agents inside the network, or outside. The two kinds of requests may merge into a single one, comprised of *old* (i.e. sent by neighbors inside the network) and *new* (i.e sent by customers) requests stored in the same place.

Moreover, within the limits of some predefined dynamic capacity, agent i can receive simultaneous requests from several neighbors, or store (commit to) several consecutive requests from the same neighbor. This capacity can be decomposed in as many sub-capacities as i's neighbors. Sub-capacities may differ with the neighbor, if agent i shows preferences toward some neighbors, either because it is more fitted to their requests, or because it displays more willingness to respond to their requests.
We shall consider here only state-independent priorities, which should not systematically favor requests emanating from some neighbors for two reasons at least. First, analysis of communication networks teaches that, regarding traffic, homogeneous structures display a better behavior.. Second, the specific nature of agent i's relationship with some of its neighbors, has already been taken into account at both the level where i accepts or turns down a request, and the level of the sub-capacity size.

Things may sound a little different, when considering priorities between old and new requests. Most classical methods for preventing congestion in communication networks limit the volume of data inside the network. Priority is given to transmission of "old" data . The same principle may be applied here with requests. Implementation can be achieved by gathering each agent's tasks performed in four-step cycles :

- step 1 : new requests are possibly accepted inside the conversational network ;
- step 2 : existing requests are prepared for emission;
- step 3 : new requests are prepared for emission;
- step 4 : requests are sent to adjacent agents.

Requests limitation will be obtained by blocking new requests when necessary.
Given the above assumptions, we can associate with every ordered pair of agents (i,j), a dibit ($A_{ij}(t)$,$B_{ij}(t)$), where $A_{ij}(t)$ and $B_{ij}(t)$ validate (value 1) or inhibit (value 0) *requests* from i to j, and *commitments* by j to a request sent by i, respectively. If i and j are not adjacent, we consider that emission and reception are validated, and hence set the dibit value to (1,1). Thus, the state of the agents network can be represented by a square binary matrix.

At each tick of the clock, this matrix gives a global view of the interactions between agents : hence, it represents the state of the network.
We assume moreover that the conversational network is controlled by a centralized algorithm, requiring synchronous transmission toward a control center, of data relative to agents states, in a such a way that :

- during data transmission, no request or commitment is allowed;
- information sent to the control center is sufficient to update the state matrix.

The state matrix being boolean and finite, the network states set is also finite. We can therefore apply finite state machine concepts as developed in communication network architectures, and consider for instance that the conversational network has two fundamental functions : emitting requests and emitting commitments.

Therefore, the state can be considered as the matrix of a game of deterrence, with two abstract players, the request function (R) and the commitment function (C), both performed under the control of the control center, and agents set as strategic sets.
Each row i (resp. each column j) of the matrix represents a strategy r_i of R, (resp. strategy c_j of C) corresponding to emission of a request by agent i (resp. emission of a commitment by agent j). $(A_{ij}(t),B_{ij}(t))$ is the outcome vector associated with the selection of strategic pair (r_i,c_j) at time t.
The strategies playability properties can be reinterpreted in the context of the network. Thus, a strategy r_i is :

- safe, if requests from i toward all other agents are validated;
- dangerous, if there is at least one agent j toward which request is not validated.

We define similarly safe and dangerous commitment strategies c_j.
Concepts of positive playability, playability by default, playability, and deterrence are then directly applicable.
Assume for instance that request r_i is deterrent vis-à-vis commitment c_j. The usual conditions can be interpreted here as follows :
1) agent i can emit a request;
2) agent j cannot commit to a request sent by agent i;
3) there is another agent j' which can commit to a request sent by agent i.

To focus on relations between adjacent agents, we shall associate with each pair of non-adjacent agents, the outcome pair (1,1) : thus, regarding strategies properties, requests and commitments between non-adjacent agents are irrelevant.
Although not compulsory, the game can be played each time the state matrix is modified, that is at each tick of the clock indicating the beginning of a new cycle of tasks. So, network control is done through a multi-stage game.

5 Congestion Control through Games of Deterrence

5.1 Congestion Control Algorithm

Congestion occurs when no agent is willing or able to commit. But this is not a sufficient condition, since if such a situation occurs, it does not imply that the same will go for all later instants. So, we shall define congestion by a sequence of two consecutive and identical state matrices, for which no strategy of (player) Commitment is positively playable : the network has then reached a state of deadlock.

Congestion control is done through the following algorithm, where c_j is a commitment strategy :

(1) If c_j is safe, agent j can commit to every request, whether new, or old;

(2) If c_j is not playable, j does not commit to any request;

(3) If c_j is playable by default, j commits only to old requests .

On the whole, the algorithm enables the conversational network to get rid of old requests, before allowing new ones in the network.

Let us then consider a state of the network at time t, for which all commitment strategies are safe, and suppose that at time t+1, the commitment sub-capacity C_{ij} of a agent i associated with direction j is saturated. The request sub-capacity R_i of i is empty. Indeed, were it is not the case, then at time t, at least one neighbor k of i would have been unable to commit, and hence commitment strategy c_k would not be safe, which contradicts the assumption. So R_i is empty, and at time t+2, C_{ij} will again accept requests from agent j.

It follows that there can be no direct transition between a state of the network for which all commitment strategies are safe, and congestion. The conclusion extends with no difficulty to the case where at time t, only one commitment strategy c_j was safe. Indeed, at time t+1, there is at least one adjacent agent j of i, such that request sub-capacity R_j is not saturated.

Hence, *congestion can only occur when commitment strategies are playable by default*. Let us then assume that in the state prior to congestion, there was only one agent i willing to commit, say to a request of agent j. Applying the algorithm creates a new possibility of request for j. In the next state, preparation of requests for emission will move the possibility backward, toward one of j's commitment sub-capacities, and so on. The same applies when there is more than one request or commitment possibility. Hence applying the algorithm prevents congestion.

5.2 Congestion and the Multiplicity of Game Solutions

The above discussion carried the implicit assumption that the state of congestion derives straightforwardly from the resolution of the game. But, as seen in example 1, the game of deterrence might have several solutions, for instance, one where C has some positively playable strategy, and another where its strategies are playable by

default. Should we then rely on the first solution and consider there is no congestion, or should we consider the second solution and "declare" the state of congestion ?
Solving this question obviously requires revisiting the notion of congestion in the light of the possible multiplicity of solutions of the playability system.

1) It is clear that in the case where no solution of the playability system is comprised of strategies playable by default, there is no congestion.
2) Things are a little more delicate when such a solution exists, and we have then to consider two different sub-cases :
- the solution is unique, and then clearly congestion occurs, and the control algorithm can be triggered;
- there is at least another solution, including a positively playable strategic pair.

In the second sub-case, let us consider for instance a strategy i of Request and a strategy j of Commitment, such that (i,j) is playable by default in solution 1 and positively playable in solution 2;

(i,j)'s playability by default in solution 1 means that neither i nor j are safe, and hence there exists a pair (i',j'), playable in solution 1, and such that implementation of (i,j') leads to an unacceptable outcome for Request, while implementation of (i',j) leads to an unacceptable outcome for Commitment.

Positive playability of (i,j) in solution 2, implies that (i',j') is not playable. So :
- since agent j' cannot commit, agent i can emit a request which will possibly be responded to by another agent, for instance j;
- since agent i' cannot request, agent j can commit to the request of i for instance.

Consequently :
- agent j' will not commit, if there is an agent k able to send j' a request that j' wants to turn down. For instance, j' may pretend to be busy in order to turn down k's request. It would then be difficult for j to accept another request.
- Similarly, agent i' cannot send a request, if there is an agent l, which can commit to that request, while i' does not want l to do the job.

Clearly, solution 2 leads to absence of congestion.
All what is left is some kind of focal-point analysis, to see whether one can ensure selection of solution 2.
If the situation is to be characterized as one of no congestion, it stems from the model structure that decisions are decentralized, and hence agent i, for instance, has no control on what agent j does. In the case under consideration, this raises no difficulty, since whatever the solution, each strategy is always playable, either by default or positively, and hence can be played in both cases. The difference lies in the environment in which these strategies are played. In solution 1, by application of the control algorithm, no request is accepted from outside the network, while this is not the case in solution 2 where the algorithm is not applied.

Incoming requests are (possibly) turned down by the management. So if information on the state of the agents is provided in due time, the management can consider that everything is *as if* solution 2 had been selected.
At first sight, the case where a strategy, say i', is not playable in solution 2 seems more difficult. Indeed, there is here a tangible difference in terms of actions taken by agent i' in both solutions. But given that only one strategic pair is to be selected, non

playability of i' in solution 2, implies the existence of an alternative strategy, say i positively playable, and here we go again.

On the whole, if the solution set of the playability system includes a solution with strategies playable by default, and another solution with at least one positively playable equilibrium, congestion needs not to be declared, and the control algorithm needs not to be triggered.

5.3 Example 1 Revisited

Let us consider an organization comprised of four agents, 1,2,3, and 4, and let us assume that the state of the conversational network is represented by the matrix of example 1 :

Table 2. Example 1 revisited

Commit

Request		c_1	c_2	c_3	c_4
	r_1	(1,1)	(0,0)	(1,1)	(0,0)
	r_2	(1,0)	(1,1)	(1,1)	(0,0)
	r_3	(0,1)	(0,0)	(1,1)	(1,1)
	r_4	(1,0)	(1,0)	(1,0)	(1,1)

We know the game has two solutions :

1) $\{J(r_1) = 0; J(r_2) = 0; J(r_3) = 0; J(r_4) = 1; J(c_1) = 0; J(c_2) = 0; J(c_3) = 0; J(c_4) = 0\}$.
2) $\{J(r_1) = 0; J(r_2) = 0; J(r_3) = 1; J(r_4) = 1; J(c_1) = 0; J(c_2) = 0; J(c_3) = 0; J(c_4) = 1\}$;

The first one characterizes congestion, since Commit has no playable solution.
The second one does not, since c_4 is playable.
It stems from the above discussion that congestion will then not be declared.
Though the second solution leads to two positively playable equilibria (r_3,c_4) and (r_4,c_4), only the latter should be selected, since it minimizes the risk for Request.
Selection of this equilibrium means that agent 4 who wants a task to perform will do it himself : this happens sometime too !

6 Fuzzy Games of Deterrence

6.1 Possibilistic Logic

In standard matrix games of deterrence, outcomes are binary numbers, while the playability system is defined as a logical conjunction . Hence, the solution set is a consistent set of binary values satisfying this system. As seen above, there may be several solutions. Furthermore, each solution may generate several equilibria.

Introducing a degree of truth on outcomes, or playability indices, or both, is a way to handle imperfect knowledge, which is usually referred as fuzziness.
There are a number of ways to introduce fuzzy operators, each one leading to a different possibilistic or probabilistic logic.
We shall here focus on the so called "probabilistic" definition of classical operators, computed as follows:

- conjunction $(x, y) \Rightarrow x*y$
- disjunction $(x,y) \Rightarrow (x+y - x*y)$
- negation$(x) \Rightarrow (1-x)$

We shall furthermore restrict fuzziness to playabilities indices, and extend definitions of playability and deterrence.
Considering the strategic set S of a player, we now call:
Positively playable strategy any pair $(s, J(s)) \in S \times [0,1]$, where J(s) is the degree of positive playability
Strategy playable by default any pair $(s, \hat{J}(s) \in S \times [0,1]$, where $\hat{J}(s)$ is the degree of playability by default.
Now, positive playability and playability by default may be no more mutually exclusive and in turn deterrence relation, may not be Boolean anymore.

6.2 Fuzzy Path of Deterrence

If any game of deterrence can be fuzzyfied, we shall here only discuss R-path games, which display particular properties, and the solutions of which are easy to compute.

Let us note $s_1,s_2,s_3,\ldots$the vertices of the R-path graph of deterrence
The playability system reduces to the following equations:
$J(s_1)=1$; $J(s_2)=0$; $J(s_3)= v$; $J(s_4)=(1-v)*v$; $J(s_5) =(1-v+v^2)*v$ and more generally,
$J(s_{2k-1}) =v(1+v^{2k-3})/(1+v)$ and $J(s_{2k})=v*(1-v^{2k-2})/(1+v)$ where $v= 1- \hat{J}_R$

$$\text{hence } 1-v = \prod_{k=1}^{p}(1+ v^{2k-1}) /(1+v)$$

An obvious solution corresponds to v=0 and it can be easily shown that there is another unique solution in the interval [0,1].
Figure 1 shows the degree of positive playability of strategies of odd and even rank respectively, for p varying from 3 to 10, and the corresponding plotted results.

One sees that positive playability of odd strategies decreases along the path, while on the opposite, positive playability of even strategies increases.
This means that :

- strategy s_1 acts as a "deterrent" umbrella vis-à-vis the other strategies of Row, its deterrent effect decreasing with the distance to the strategy under consideration
- the spread between playabilities of two adjacent nodes decreases along the path.

6.3 Example 2

To illustrate the above computation, let us consider the game represented in table 2.

Table 3. Example 2

Charlotte

		$c_1 = R \wedge C$	$c_2 = R \wedge \neg C$	$c_3 = \neg R \wedge C$
Raymond	$r_1 = R \wedge C$	(1,1)	(1,0)	(1,1)
	$r_2 = R \wedge \neg C$	(0,1)	(1,1)	(1,0)
	$r_3 = \neg R \wedge C$	(1,0)	(0,1)	(1,1)

The above matrix represents an interaction framework such that :

1. a state of the world where both players would neither request nor commit themselves is discarded, since then the organization would carry out no action, and furthermore, there would be no interaction between the players;
2. there are only two situations in which Raymond feels uncomfortable :
- if he requests and refuses to commit when Charlotte requests and commits;
- if he doesn't request and commits, while Charlotte requests and doesn't commit. These two situations may be interpreted as follows : when Charlotte both requests and commits, Raymond considers it unacceptable not to answer Charlotte's openness, by refusing to commit; but when Charlotte simultaneously requests and refuses to commit, Raymond considers it unacceptable to simply commit and not request, since it would mean that the two players are not on an equal foot.
3. there are three situations where Charlotte feels uncomfortable:
- the first two are similar to the two situations here above;
- in the third one, Charlotte considers it unacceptable to request and commit, while Raymond only commits: maybe is it a matter of politeness…

It can be very easily seen that the graph of deterrence is an R-path with r_1 as a root.
In the non-fuzzy game, Raymond has only one playable strategy r_1 (positively playable) while all strategies of Charlotte are playable by default.
Fuzzyfication adds another solution :
$\{J(r_1) = 1 ; J(c_2) = 0 ; J(r_3) = .48 ; J(c_1) = .25 ; J(r_2) = .36 ; J(c_3) = .31\}$

Raymond will select r_1 , that is the same strategy than in the non-fuzzy case, while Charlotte will select c_3, which leads to no contradiction with the non-fuzzy case, since in the latter all strategies are playable (by default), and hence so is c_3.
Selection of strategic pair (r_1,c_3) leads to an acceptable outcome for each player. The corresponding equilibrium can be interpreted as follows : Raymond will request while being ready to commit, which is not necessary since Charlotte only wants to commit, and hence a task is transferred from Raymond to Charlotte.

Coming back to multi-agent systems, the game here above may be interpreted as the state of a three agent conversational network. The non-fuzzy solution of the game may trigger congestion, since no agent is ready to commit .

p	v	J_1	J_3	J_5	J_7	J_9	J_{11}	J_{13}	J_{15}	J_{17}	J_{19}
3	0.48	1	0.48	0.36							
4	0.67	1	0.67	0.52	0.45						
5	0.76	1	0.76	0.62	0.54	0.49					
6	0.82	1	0.82	0.70	0.62	0.56	0.53				
7	0.86	1	0.86	0.76	0.68	0.63	0.59	0.55			
8	0.89	1	0.89	0.80	0.73	0.68	0.63	0.60	0.57		
9	0.91	1	0.91	0.83	0.77	0.72	0.68	0.64	0.62	0.59	
10	0.92	1	0.92	0.86	0.80	0.76	0.71	0.68	0.65	0.63	0.61

p	v	J_2	J_4	J_6	J_8	J_{10}	J_{12}	J_{14}	J_{16}	J_{18}	J_{20}
3	0.48	0	0.25	0.31							
4	0.67	0	0.22	0.31	0.36						
5	0.76	0	0.18	0.28	0.34	0.38					
6	0.82	0	0.15	0.24	0.31	0.36	0.39				
7	0.86	0	0.12	0.21	0.27	0.32	0.36	0.38			
8	0.89	0	0.09	0.17	0.24	0.29	0.32	0.35	0.38		
9	0.91	0	0.08	0.15	0.21	0.25	0.29	0.32	0.35	0.37	
10	0.92	0	0.07	0.13	0.18	0.22	0.26	0.29	0.32	0.34	0.36

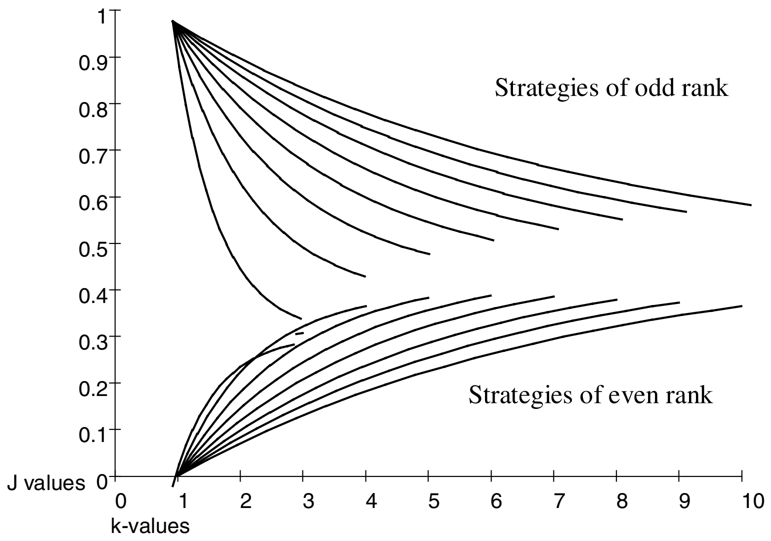


Fig.1. Degrees of playability for strategies belonging to R-paths.

On the opposite, fuzzyfication enables to avoid congestion. Indeed, it orders playabilities of Commitment's strategies. Obviously, the latter will select the strategy with the highest degree of positive playability. In other words, agent 1 will send a request, which will be responded positively by agent 3, with both agents being happy with the interaction, since they both get an acceptable outcome.

More generally, regarding congestion, the critical point, is the existence of multiple solutions with equilibria playable by default. When playability indices are no more binary, one can order the various equilibria and select the strategic pair which leads to an acceptable outcome for both agents.

7 Conclusion

Representing organizations as multi-agent systems networks, with communications modeled using games of deterrence, paves the way for a mix-management, that is :

- a decentralized management when some agents at least manage successfully to work together, and hence avoid bottlenecks;
- a centralized / hierarchical management, when bottlenecks are about to occur.

In order that the control algorithm would not be triggered by bottlenecks, but shortly before the latter occur, agents thresholds may be defined in such a way that when the algorithm is triggered, all agents have still some capacity (or sub/capacity) available.
Moreover, the threshold values can differ, not only with agents but also with sub-capacities for a given agent, thus giving the management the required flexibility to face a variety of situations, some where congestion avoidance should be prioritized, some where priority should be given to workflow maximization.
Thus the control algorithm fully prevents bottlenecks and hence rationalizes workflow.
Beyond technical refinements under current analysis, like multi-stage dynamics, priority systems, autonomy and self-regulating teams [2], the multi-agent systems approach presented in this paper, paves the way for a number of applications, among which :

- real time workflow management;
- knowledge management;
- human resources management
- organization design
- and last, but not least, e-business.

References

1. Bestougeff, H., Bouaissi, A. Multi-Agent Architecture to Model and Simulate a Business Network of Recurrent Conversations and Associated Tasks, Proceedings EMCSR-98- From Agent Theory to Agent Implementation,Vienne,(1998) Vol.2. 767-772.
2. Bestougeff, H., Self Regulating Teams and Games of Deterrence,2nd World Multiconference on Sytemics, Cybernetics and Informatics, Orlando (1998)
3. Bestougeff, H., Bouaissi, A., Dynamic BPR using a Task/ Communication Model, ACM SIGGROUP Bulletin Vol 18 N°3 (1997).
4. Bestougeff, H., Rudnianski, M., Games of Deterrence and Satisficing Models Applied to Business Process Modelling, Proceedings AAAI Spring Symposium, Stanford (1998).

5. Rudnianski, M., Deterrence, Fuzzyness and Causality, Proceedings ISAS 97, World Multiconference on Systemics, Cybernetics and Informatics, pp 473-480, Caracas (1997).
6. Rudnianski, M., Multipolar Deterrence in a Dynamic Environment, IEEE Systems, Man and Cybernetics (1995), vol 5, pp 4279 – 4285.
7. Rudnianski, M., Deterrence Typology and Nuclear Stability. A Game Theoretical Approach, in Defense Decision Making, R. Avenhaus, H. Karkar, M. Rudnianski, editors, pp 137-168, Springer Verlag, Heidelberg, (1991)
8. Rudnianski, M., d'Assignies, A., from MAD to MAD , Strategic Stability in the Post-Cold War World and the Future of Nuclear Disarmament, M.L. Best, J. Hughes-Wilson, A. Piontkowsky, editors, NATO ASI series, Disarmament Technologies, vol 3, pp 229-259, Kluwer Academic Publishers, (1995).

Web Agents Cooperating Deductively

Richard Waldinger

Artificial Intelligence Center, SRI International, Menlo Park, California, USA
waldinger@ai.sri.com
www.ai.sri.com/~waldinge

Abstract. A framework called GUIDE is presented in which many agents can cooperate to answer a query or perform a task. Agents may be heterogeneous: they need not be intended to work together and need not share any vocabulary or representational conventions. The query can be phrased without knowing which agents are available or appropriate to carry it out. Query and agents are linked by a common application-domain theory. The query is phrased as a theorem; the answer is extracted from a proof in the theory. The answer may depend on background knowledge implied by the theory. The GUIDE's approach is domain-independent but is illustrated by answering questions involving maps and directories.

1 What's the Problem

Our problem is to get software agents distributed over the web to work together to answer a query or solve a problem. Typically, the poser of the query will not know what agents are available or appropriate to work on the task. Furthermore, the agents are heterogeneous: they are generally not designed to work together and have been built with different ways of talking about the world.

Some background knowledge and ingenuity may be required to decompose and transform the query, but our emphasis in developing the GUIDE framework is on straightforward questions, not tricky puzzles. Also, although we have experimented with connecting to a spoken-language-understanding system, our main interest is in the logical aspect of the problem, once the language has been understood.

This work is still in its preliminary stages, but a GUIDE pilot implementation has been carried out. We have considered several application domains, including company employee data, a smart kitchen, and a question-answering guidebook. Our examples will come from this last domain.

1.1 Sample Queries and Agents

Here are some examples of the kinds of questions we have been considering:

How far is it in kilometers from my house to the White House?
Show me an aerial photograph of SRI?
Where can I find a wrench around here?

We assume that there is no one agent that can handle any of these problems. The agents we have been using can perform the following smaller sorts of tasks:

Given a person's name, find his or her address.
Given the name of a store or institution, find its address.
Given an address, find the corresponding latitude and longitude.
Given a latitude and longitude, show me an aerial photograph of the area.
Given two latitude/longitude pairs, find the distance between them.
Find the latitude and longitude of my GPS receiver.
Given an address, find nearby stores of a given sort.

1.2 Interoperability

Our problem, then, is to decompose and transform the queries into subqueries that are small enough for the individual agents to handle. Linking query to agents may depend on background knowledge that no agent possesses. The problem is compounded by the fact that the query may be phrased in vocabulary different from what the agents expect, and that one agent may use a different representation from another. For example, the query about the distance between houses requests its answer in kilometers; the agent that computes the distance between latitude/longitude pairs gives its answer in miles. Another example: the agent that computes the latitude and longitude of an address gives answers in *compass* notation, such as 37.457056N and 122.17795W, where the sign is indicated by letters (N, S, E, and W); the agent that computes the distance between two latitude/longitude pairs expects input in *sign notation*, such as 37.457056 and -122.17795, where the negative sign is indicated by "-". While we assume there are agents or programs that can do these conversions, part of our problem will be to incorporate these conversion agents appropriately into the solution of the problem.

2 Our Approach

Our approach is based on automated deduction and program synthesis (e.g. [1], [2]).

- We formulate an axiomatic theory of the application domain rich enough to express the properties of the constructs in our queries and agents. Background knowledge necessary for deducing answers to queries is also included in the application-domain theory.
- We associate with each agent some axioms that advertise its capabilities. These axioms describe the services that the agent provides, and are included in the application-domain theory.
- We express the query as a theorem, expressed in the language of the application-domain theory. Typically, the theorem will express the existence of the entity that the query is asking us to find: a distance, a picture, an address.

- We prove that the theorem follows from the axioms of the application-domain theory, using an automatic theorem prover. The proof is restricted to be sufficiently constructive so that, in proving the existence of a desired entity, we are forced to indicate a method of finding it.
- Certain symbols of the theory are attached to agents; this means that when such a symbol appears in an axiom used in the proof, the corresponding agent will be invoked. In particular, if the method of finding the desired entity requires information from certain agents, those agents will be invoked while the proof is in progress.
- We extract the desired entity from the proof of the theorem.

If more than one answer is required, we may find other proofs of the same theorem that yield different answers, until the search space or our time limit is exhausted.

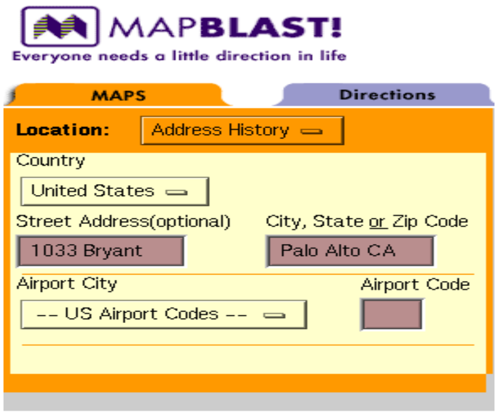
This will be illustrated in more detail in the balance of this paper.

3 Advertisement for an Agent

Many of the agents we have considered are web agents, invoked via web pages.

3.1 A Web Page

As an example of an agent that corresponds to a web page (Figure 1), let us consider the agent that finds the latitude and longitude of a given street address. This agent corresponds to one of the capabilities of the web page Mapblast`http://www.mapblast.com`). Figure 2 shows the page that Mapblast generates when it shows the map and computes the latitude and longitude.



Location: Address History

Country
United States

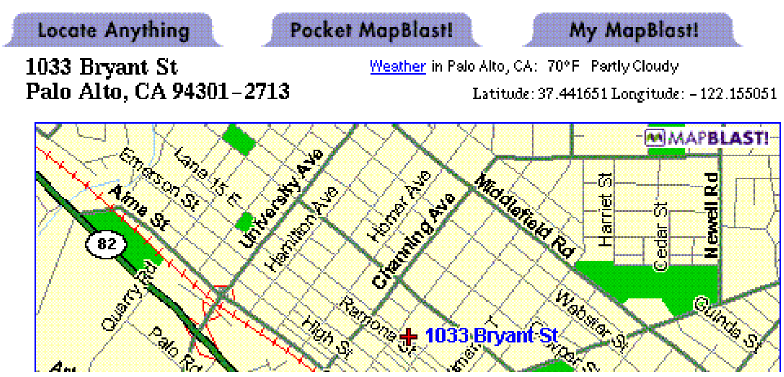
Street Address(optional): 1033 Bryant

City, State or Zip Code: Palo Alto CA

Airport City: -- US Airport Codes --

Airport Code:

Fig. 1. Mapblast requests an address



1033 Bryant St
Palo Alto, CA 94301-2713

Weather in Palo Alto, CA: 70°F Partly Cloudy
Latitude: 37.441651 Longitude: -122.155051

Fig. 2. Mapblast produces a map and a latitude and longitude

3.2 Turning a Web Page into an Agent

Web pages are not themselves agents, but we can turn a web page into an agent that can be invoked during a proof, via a three-stage process:

- Build a program in the language WEBL [3] that can take as inputs the information the web page requests and return as outputs the information the web page yields. This converts the web page into a procedure.
- Attach the WEBL procedure to a symbol in the application-domain theory, so that the procedure can be invoked when that symbol plays a role in the search for a proof. Our method of doing this uses SRI's Open Agent Architecture (OAA) [4].
- Introduce an axiom that advertises the capabilities of the procedure, and relates it to the concepts of the application-domain theory. In that way, the procedure will be invoked when it is appropriate to the task in hand.

Together, the web page, WEBL program, and advertisement can be regarded as an agent, which waits to be invoked until its services are required.

3.3 A WebL Program

For example, here is a portion of the WEBL program corresponding to the Mapblast web page:

```
// if zip specified, good enough.  Otherwise, try city,state
if (zip == "") and (city != "") and (state != "") then
   zip = city + "," + state
end;

// need at least a zip
if zip == "" then
   return "[]"
end;
```

```
PrintLn("Looking up lat/long for: " + street + " " + zip);
try
   P = Timeout(60000, Retry(
                              // query site with correct params
              PostURL("http://www.mapblast.com/mblast/map.mb",
                  [.  loc="us", CMD="GEO",AD2=street,AD3=zip .]
       )));
catch E
   on true do
      PrintLn(E);  // Print Error
      return "[]"  // Return failure: couldn't load page
end;
```

3.4 Advertisement for a Web Agent

Here is the axiom that advertises the Address-to-Lat-Long agent:

```
(=>
  (address-to-lat-long-agent
     ?street ?city ?state ?country ?lat ?long)
  (= (address-to-lat-long ?street ?city ?state ?country)
     (lat-long-sign ?lat ?long)))
```

Here, `address-to-lat-long-agent` is a predicate symbol in the logic that is attached procedurally to the WEBL program for Mapblast. The axiom means that, given address strings `?street`, `?city`, `?state`, `?country`, the agent will yield strings `?lat` and `?long` corresponding to the latitude and longitude of the place at the corresponding address. (The question-mark prefix `?` indicates a variable. Variables in axioms are taken to be universally quantified; in other words, the above axiom should be read "For all strings `?street`, `?city`, `?state`, `?country`, ...") The procedural attachment gives the effect of adding a large number of axioms to the application-domain theory, giving the latitude and longitude for every address in the Mapblast database.

The domain-theory function `address-to-lat-long` takes the same address strings and yields an abstract latitude/longitude pair. The function `lat-long-sign` takes numerical strings `?lat` and `?long` and yields the corresponding abstract latitude/longitude pair. These strings are positive and negative real numbers. Another function, `(lat-long-compass ?lat ?long)`, also yields an abstract latitude/longitude pair from strings `?lat` and `?long`, but it requires that `?lat` and `?long` be in compass notation, such as 37.457056N and 122.17795W. The abstract latitude/longitude pair is independent of notation.

While the domain-theory functions, such as `address-to-lat-long`, are independent of any agent, the meaning of the procedurally linked predicate symbol `address-to-lat-long-agent` is directly tied to the performance of the WEBL program for Mapblast. In particular, should the agent be unavailable or too slow, the antecedent

```
(address-to-lat-long-agent
   ?street ?city ?state ?country ?lat ?long)
```

will be false. The theorem prover will need to find another agent that can provide the same information, or else answer its query in some other way.

Here is another axiom, which advertises the capabilities of the agent that can compute the distances between two latitude/longitude pairs:

```
(=>
  (lat-long-to-distance-agent ?lat1 ?long1 ?lat2 ?long2 ?real)
  (= (distance-in-miles ?real)
     (distance-between
       (lat-long-to-place (lat-long-compass ?lat1 ?long1))
       (lat-long-to-place (lat-long-compass ?lat2 ?long2)))))
```

Here `lat-long-compass` is the domain-theory function that takes two strings that stand for latitude and longitude in compass notation (e.g., 37.457056N) and yields the corresponding abstract latitude/longitude pair. The function `lat-long-to-place` takes an abstract latitude/longitude pair and yields the corresponding abstract place. The function `distance-between` takes two places and yields the abstract distance between them. Abstract distances are independent of any system of units. The function `distance-in-miles` takes a number, interprets it as a distance in miles, and yields the corresponding abstract distance. Thus

```
(= (distance-in-miles .62)
   (distance-in-kilometers 1))
```

Online documents other than web pages can be converted to procedures by WEBL programs. While our WEBL agents have been interpreting pages in HTML, WEBL is also applicable to pages annotated in XML.

Turning web pages into agents in this way is a labor-intensive process. There is, however, an effort to develop a new annotation language, the DARPA Agent Markup Language (DAML), especially to annotate web pages so they may be invoked as agents. It is hoped that DAML-annotated web pages will not need to have corresponding WEBL programs. Furthermore, we may expect that axioms to advertise the capabilities of the agent will be generated automatically from the DAML page.

3.5 Turning Procedures into Agents

While many of our agents are obtained from web pages, others are local database systems or simple functional programs. In each case, the principle is the same; we introduce axioms that describe their behavior and advertise their capabilities. These advertisements determine when the agents are invoked. Symbols in the advertisement axioms are linked to the procedures in question.

For example, two LISP procedures take the numerical notation for latitude and longitude, as signed real numbers, and convert them into their respective compass notations. These are advertised by the axiom

```
(= (lat-long-compass
      (latitude-sign-to-compass ?lat)
      (longitude-sign-to-compass ?long))
    (lat-long-sign ?lat ?long))
```

Here `?lat` and `?long` are strings that represent the latitude and longitude, respectively, as signed real numbers. The function symbols `latitude-sign-to-compass` and `longitude-sign-to-compass` are attached to the two procedures that perform the conversions.

The relationship between miles and kilometers is described by the axiom

```
(= (distance-in-kilometers ?real)
    (distance-in-miles (* ?real .62)))
```

Because the function symbol `*` is attached to a procedure that can perform multiplication, this axiom can be regarded as the advertisement for an agent that can convert from kilometers to miles.

One could have a separate axiom that performs the conversion in the other direction. It is more economical, however, to introduce the division agent, which is the inverse of the multiplication agent, by the axiom

```
(=>
   (not (= ?real2 0))
   (= ?real1 (* ?real2 (/ ?real1 ?real2))))
```

The function symbol `/` is attached to a procedure that performs division. Together, the above two axioms allow the theorem prover to convert between miles and kilometers in either direction.

4 Queries

For many of the applications we are considering, such as a smart kitchen or a trip guide, it is natural to deal with queries in speech or natural language, or selected from a menu. In this work, however, we have been considering queries in a logical form. It is assumed that whatever query method is appropriate, it will be possible to translate those queries into a logical form.

In fact, the SRI Artificial Intelligence Center Natural Language Understanding Program has developed a system called Gemini [5] that can translate speech into a logical form. Although the Gemini logical form is different from the one presented here, we have extended the domain theory to answer some questions phrased in the Gemini logical form as well.

I can express the query How far is it from my house to the White House? in logical form as

```
(find ?real
 (= (distance-in-miles ?real)
    (distance-between
      (address-to-place (name-to-address ‘‘Richard Waldinger’’))
      (name-to-place ‘‘The White House’’))))).
```

The construct `(find ?real ...)` can be thought of as a constructive existential quantifier. More precisely, it means that we are to prove the existence of a quantity `?real` satisfying the indicated relationship, where the proof is to be restricted to indicate a method for finding that quantity.

This representation is simplified in that it assumes that Richard Waldinger and the White House are unique and unambiguous. Otherwise, we would have to use a relation instead of a function for `name-to-address` or supply additional information to make the identities apparent.

Note that the query does not dictate any particular method of solution. If there were an agent that could consult a table of distances between all people's houses and public landmarks, it could solve the problem immediately. Otherwise, the theorem prover will need to use axioms from the domain theory to decompose the query into tractable subqueries.

To ask Where can I find a wrench around here?, we can formulate a query

```
(find ?street
  (and
    (instance-of ?object wrench)
    (location-of ?object ?place)
    (< (distance-in-miles
         (distance-between here ?place))
       1)
    (= ?place
       (address-to-place ?street ?city ?state ?country)))).
```

In other words, we want to show the existence of an object in the class of wrenches that is located in a place that is within a mile from here and we want to find the street address of that place.

To find an aerial photograph of SRI, we formulate the query

```
(find ?picture
  (and
    (instance-of ?picture aerial-photograph)
    (subject-of
      ?picture
      (name-to-place ‘‘SRI International’’)))).
```

In each case, the query does not indicate a particular method of solution or point to appropriate agents; using axioms from the domain theory, the query must be decomposed and transformed into subqueries that can be handled by the agents currently at our disposal.

This approach is robust with respect to the introduction and removal of new agents. If a new agent is added, its advertisement axioms are introduced into the application-domain theory; if an agent is removed, its axioms may be removed as well. No agent needs to know what other agents are available; agents are invoked by tasks and services, not by name. Thus the same query may be answered differently at different times, depending on what agents are available.

At this point, it may be useful to see some of the axioms from the application-domain theory.

5 The Application-Domain Theory

The application-domain theory must have enough information to define all the constructs that appear in queries and in agent advertisements. If the answer to a query depends on a relationship between some of these constructs, that relationship must be implied by the domain theory as well.

For the queries about distances we have been considering, there are axioms that relate the various constructs we have introduced. For example, the following axiom relates distances to latitude/longitude pairs:

```
(= (distance-between ?place1 ?place2)
   (lat-long-to-distance
     (place-to-lat-long ?place1)
     (place-to-lat-long ?place2)))
```

This axiom suggests that one way to find the distance between two places is to find the latitude and longitude of the places, and then to find the distance between the two latitude/longitude pairs.

The following axiom relates addresses and places with latitude/longitude pairs:

```
(= (address-to-place ?address)
   (lat-long-to-place (address-to-lat-long ?address)))
```

Some axioms express general world knowledge, such as that hardware can be found in hardware stores and that wrenches are hardware.

```
(=>
  (instance-of ?place hardware-store)
  (exists (?object)
    (and
      (instance-of ?object hardware)
      (location-of ?object ?place))))
```

and

```
(subclass-of wrench hardware)
```

This is of course a first approximation. A hardware store could be out of wrenches; if the store has one, it must be purchased but may be kept indefinitely. Wrenches may also be found in gas stations and in people's houses, but they might not permit you to borrow one; if they do, it need not be purchased but must eventually be returned. Such nuances are not reflected in our current domain theory.

Using axioms such as these in a proof allows us to compose answers that depend on background knowledge that may not occur explicitly in any of the agents under consideration. For instance, the directory that gives us the location of a hardware store may fail to mention that the hardware store sells wrenches.

Constructing a large application-domain theory is a massive effort, but one that many projects can take advantage of. We are building the GUIDE theory on top of other similar efforts, at SRI and elsewhere, including

Artifact Theory: An SRI effort [6] to build a formal theory of physical agents that will take account of space, time, causality, processes, and other more specific concepts.

Rapid Knowledge Formation: A DARPA effort [7] to build a framework by which many subject-matter experts can collaborate to build a formalization of a piece of scientific knowledge.

Cyc: A long-term effort [8] to build a large formal repository of commonsense world knowledge.

6 The Snark Theorem Prover

The approach we have been following relies heavily on strong theorem-proving technology. We have been using the theorem-proving system SNARK [9] for this purpose; SNARK is a first-order logic system implemented at SRI by Mark Stickel. Although perhaps many theorem provers are applicable for our purpose, SNARK is especially appropriate because of the following characteristics:

- As we anticipate invoking large numbers of agents, and relying on large stores of background knowledge, the theorem prover must be able to deal with large theories. Snark has good indexing methods and domain-specific heuristic methods for restricting search.
- We need to deal well with equality. Snark has capabilities for paramodulation and fast rewriting.
- We must be able to deal with taxonomic knowledge efficiently, such as "because wrenches are hardware, any property of hardware also holds for wrenches." Snark's sort mechanism can do fast inferences about taxonomic hierarchies.
- Snark has a capability for restricting proofs to be constructive and for extracting answers to queries for proofs.
- We must have a procedural-attachment mechanism by which function and predicate symbols in the logic can be associated with procedures that can be computed while the proof is under way. Snark's procedural-attachment mechanism allows the invocation of arbitrary LISP programs and has been connected to SRI's Open Agent Architecture, so that programs and agents may be invoked as the proof is in progress.

Snark is the product of many years of research in theorem proving. It has been used in SRI's High Performance Knowledge Base (HPKB) project [10], which answered questions about historical, economic, geographic, and military aspects of the Persian Gulf, and which required thousands of axioms. It was also used in the Amphion system [11] of NASA; this system composes software from subroutine libraries to meet a user's specifications. It has successfully composed programs of dozens of lines, entirely automatically.

SNARK can be tuned to exhibit high performance in a particular application. One such method is a symbol ordering, in which SNARK is told to prefer one symbol over another. This gives it a sense of direction in applying rules. For this application, we have it prefer symbols that are more concrete over those that are abstract. For instance, symbols that are used in queries are the most abstract, like `distance-between`. Symbols that are attached to procedures are most concrete, like `lat-long-to-distance-agent`. Some symbols are intermediate, like `lat-long-to-distance`. For queries we have considered, SNARK requires only a few seconds; the time to answer the query is dominated by the time to get a response from web pages. Of course, up to now, we have been using only the agents that are available in the OAA library—not thousands or millions of agents.

Although the correctness of the answer provided by the GUIDE is no better than the accuracy of the information provided by the agents and the axioms of the application-domain theory, the proof provided by SNARK establishes that the answer does correctly follow from that information.

7 Related Work

This work is built on top of SRI's Open Agent Architecture and augments its capabilities. OAA does have Prolog-like reasoning ability and allows agents to be invoked by predicate symbols, but it does not have a full-fledged theorem prover; for example, it has neither a sort structure for taxonomic reasoning nor a capability for reasoning about equality.

The Infomaster system [12] also uses reasoning to answer queries based on web agents. Its emphasis is on looking up answers in multiple databases, and it has been applied to searching catalogs and housing advertisements. It has some theorem-proving ability, but, again, no sort structure and no equality reasoning.

The current system may be regarded as a variation of Amphion, which we referred to earlier. While Amphion constructs a program that can answer many instances of the same query, the GUIDE answers individual queries directly. While Amphion constructs programs by composing subroutines, the GUIDE constructs answers by composing information from agents.

We have already emphasized our dependence on WEBL and, eventually DAML, for allowing us to give semantic import to information provided by web pages. We also intend to use SRI's spoken-language understanding system Gemini, so that queries, and perhaps even axioms, may be expressed in natural language.

8 Summary

In the GUIDE framework, multiple disparate agents can cooperate to solve common tasks, even though they have not been designed to work together. Procedures are regarded as agents by annotating them with axioms that advertise their capabilities; the procedures are linked to symbols in the axioms, so that if an attempt is made to use the axiom in a proof, the corresponding procedure will be invoked as the proof is under way. Web pages are treated as procedures by annotating them with WEBL programs, but other procedures can also be regarded as agents. Ultimately, we hope that DAML-annotated web pages can be transformed into agents automatically.

Axioms that describe the symbols in our axioms or queries, advertise the capabilities of agents, or provide background knowledge necessary to deduce answers to queries, define the application-domain theory. Queries are phrased as theorems in logic and proved within the theory by the theorem prover SNARK; an answer to the query is extracted from the proof. The proof provides a limited guarantee of the correctness of the answer.

In time we plan to use the spoken-language understanding system GEMINI to translate natural-language queries into logical theorems; a preliminary experiment in this direction has been successful. The GUIDE has been used experimentally as a geographical reference and travel guide, a smart-kitchen facilitator, and an employee query system.

Acknowledgments. Thanks to Mark Stickel with help in the use of the SNARK theorem-proving system. Mark also contributed to the choice of representation.

Thanks to Adam Cheyer, David Martin, and Didier Guzzoni for help in using the Open Agent Architecture. David assisted also with the OAA LISP interface, and Didier developed and altered WEBL programs to assist this work.

Thanks to Vinay Chaudhri for help with and discussions about knowledge-representation issues.

Thanks to Jerry Hobbs and Mark Gawron for help leading toward connecting the GUIDE to the GEMINI spoken-language understanding system.

Thanks to Lee Iverson, Jason Jenkins and Mabry Tyson for help in preparing this document.

The research reported here has been partly supported by DARPA under Contracts N66001-97-C-8550 (HPKB) and N66001-97-8551-00-SC-01, Subcontract PSRW-97-8551-00-SC-01 (Genoa).

References

1. Green, C. C.: Application of Theorem Proving to Problem Solving. Proceedings of the International Joint Conference on Artificial Intelligence (May 1969) 219–239
2. Manna, Z., and Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Transactions on Programming, Languages, and Systems. **2** (1980) 90–121
3. Marais, Hannes: WEBL—A Programming Language for the Web. Compaq Systems Research Center (1999) `research.compaq.com/SRC/WebL/`, temporarily unavailable
4. Martin, D. L., Cheyer, A. J., and Moran, D. B.: The Open Agent Architecture: A framework for building distributed software systems. Applied Artificial Intelligence **13** (January-March 1999) 91–128

5. Dowding, J., Gawron, J. M., Appelt, D., Bear, J., Cherny, L., Moore, R., and Moran, D.: GEMINI: A natural language system for spoken-language understanding. Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics (June 1993) 54–61
6. Hobbs, J., and Lincoln, P.: Artifact Theory (unpublished transparencies). Artificial Intelligence Center and Computer Science Laboratory, SRI International (October 1999)
7. Teknowledge: Rapid Knowledge Formation. `http://reliant.teknowledge.com/RKF`
8. Lenat, D. B., and Guha, R. V.: Enabling Agents to Work Together. Communications of the ACM **37** 7 (July 1994)
9. Stickel, M., Waldinger, R., Chaudhri, V.: A Guide to Snark. SRI International Artificial Intelligence Center (2000) `http://www.ai.sri.com/hpkb/snark/tutorial/tutorial.html`
10. Chaudhri, V. K., Stickel, M. E., Thomere, J. F., and Waldinger, R. J.: Reusing Prior Knowledge: Problems and solutions. Proceedings of the National Conference on Artificial Intelligence (July 2000)
11. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I.: Deductive Composition of Astronomical Software from Subroutine Libraries. In: Bundy. A. (ed.): 12th Conference on Automated Deduction, Nancy, France, June 28-July 1, 1994; Automated Deduction. Lecture Notes in Computer Science, Vol. 814. Springer-Verlag, Berlin Heidelberg New York (1994)
12. Genesereth, M. R., Keller, A. M., and Duschka, O. M.: Infomaster: An information integration system. SIGMOD Record (ACM Special Interest Group on Management of Data) **26** (1997) 539–542

Formal Specification of Agent Behaviour through Environment Scenarios

Hong Zhu

School of Computing and Mathematical Sciences, Oxford Brookes University
Gipsy Lane, Headington, Oxford, OX3 0BP, England
hzhu@brookes.ac.uk, Tel:++44 1865 483670, Fax:483666

Abstract. Scenarios are typical situations that may occur in the operation of a software system. Scenario analysis plays an important role in software requirements analysis and design. This paper introduces a formal notation for the specification of scenarios in agent-based systems. The autonomous and collaborative behavior of an agent can be formally specified by a set of rules describing its activity in various scenarios. The power of the approach is illustrated by the formal specification of Maes' personal assistant agent Maxims. The paper also shows that agents' social behavior, such as speech act, can also be formally specified as scenario-reaction rules.

1 Introduction

Being autonomous, proactive and adaptive, an agent-based system may demonstrate emergent behaviours, which are neither designed nor expected by the developers or users of the system. Whether or not such emergent behaviours are advantageous, methods for the specification of agent behaviours should be developed to enable software engineers to analyse the behaviour of agent-based systems. The past few years have seen a rapid growth in the research on formal models of agent-based systems specified in various formalisms and logics, such as temporal logics, first order logics, and game theories, etc., see e.g. [1, 2, 3]. However, there are few researches on the language facilities and features that support the formal specification and verification of agent-based systems, although efforts have been made to define new formal specification languages, for example, L_W [4], DESIRE [5] and ETL [6].

This paper reports our research in progress on a formal method for the specification, proof and analysis of the behavior of agent-based systems. The basic idea is to specify an agent's behaviour by a set of rules that govern its reactions to various scenarios in its environment. A formal specification language, called SLAB, is being designed to facilitate such formal specifications and analysis of multi-agent systems. Instead of giving a complete definition of the language, this paper presents the language facilities that we identified and the rationale behind the design decisions. We also illustrate by examples the use of the facilities in the specification of autonomous and collaborative behaviors of multi-agent systems.

2 SLAB – A Formal Specification Language of Agent Behaviour

It is widely recognised that formal methods are difficult to scale up. Modularity and composibility are among the solutions to this problem advanced in the literature. The question is then what are the 'modules' in a formal specification of an agent-based system and how to compose them together into a system. In search for an answer to these questions, we turned to more fundamental questions like what is the essence of agent-based computing and what makes agents an appealing and powerful approach. As Jennings pointed out [7], such questions can be tackled from many different perspectives ranging from the philosophical to the pragmatic. In the design of the SLAB language, we have taken a pragmatic approach in order to obtain a practical solution. We are concerned with the language facilities that support the specification and reasoning of agent-based systems from a software engineering point of view. In this section, we discuss how such a view led to our design decisions.

2.1 Agents as Encapsulations of Behaviour

Of course, agent is the most basic concept of agent-oriented or agent-based computing. Although there is much debate about exactly what constitute agenthood, we believe that Jennings' definition represents a common view from many researchers. It reads 'an agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives' [7]. According to this definition, an agent is an entity that observes its environment and takes its action (or reaction) according to its internal state, which can be driven by its believe, desire and intention and to follow a plan to achieve a specific goal. Thus, agents are active and persistent. They execute concurrently and autonomously.

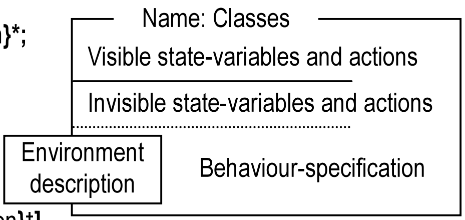
Therefore, an agent has a set of variables represents its internal state and a set of actions it can take. Being an active entity, an agent's internal state is persistent in the sense that there is always a value bound to the variables. In contrast, actions are temporal in the sense that it is possible that no action is taken at a particular time. Another difference between state variables and actions is that state variables can vary continuously as time changes. In other words, a state variable can be a function of time. To communicate with the outside world, an agent must have some actions that are observable by other agents, and be able to show its state to the outside world. For example, Maes' Maxims agents have facial expressions to communicate with the users. However, an agent also needs to be able to hide some of its internal state and actions. The state variables and actions are, therefore, divided into two types, those visible from the outside and those internal and invisible from the outside.

The most important feature that distinct agents from objects in the object-oriented paradigm is the so-called autonomous behaviour. Although an object also has internal states and a set of actions (which are called methods in object-oriented terminology), it has no control over whether and when to take an action. A method must be executed when a message is received. In contrast, an agent can decide whether or not and when to take an action when a message is received or a certain event happens in the environment. Its capability of controlling over its internal state and action is the characteristics of autonomous behaviour. In other words, an agent's behaviour is determined by its design rather than by the environment. In this sense, we say that an agent is an encapsulation of states, action and behaviour, or shortly, an encapsulation of behaviour.

We believe that the power of agent-oriented approach comes from the encapsulation of behaviour, which enable an agent to achieve its design objectives in dynamic and unknown environment by controlling its behavior and adapting its behavior according to the environment, rather than controlled by the environment. Therefore, in the design of an agent, the complexity of the problem due to the dynamic nature and unknown characteristics of the environment can be reduced to the minimum. This understanding of the concept of agent led us to the first design decision in the development of the formal specification language SLAB. That is, the basic building block in an agent-oriented specification language should be agent, which encapsulates four interrelated parts: (1) the specification of state space, (2) the specification of actions, (3) the specification of behaviour, and (4) the specification of the part of environment that it observes.

The following gives SLAB's syntax in EBNF of specifications of agents. It can also be equivalently represented in a graphic form similar to the schema in Z [8].

```
agent-description ::=
 agent name [: { class-name,}] {instantiation}*;
     [ environment-description; ]
     [ structure-description; ]
     [ behavior-description ]
 end name
structure-description ::=
 [ Var {[ * ] identifier: type; }+ ] [Action {action}+]
action ::= [*] identifier; | identifier ({ [parameter:] type,}+)
```


Name: Classes
Visible state-variables and actions
Invisible state-variables and actions
Environment description
Behaviour-specification

In SLAB, the state space of an agent is described by a set of variables with keyword VAR. The set of actions is described by a set of identifiers with keyword ACTION. An action can have a number of parameters. An asterisk before the identifier indicates invisible variables and actions.

The power of agent-based system can be best demonstrated in a dynamic environment [9, 10] because an agent can adapt its behaviour into the environment to achieve its designed purpose. Therefore, the specification of an agent-based system must also specify how the environment affects the behaviour of the agent. To do so, we must first answer the question what is the environment of an agent. A simple answer to this question is that in a multi-agent system, the environment of an agent consists of a number of agents and a number of objects. However, having defined agents as encapsulations of behaviours, we regard object as a degenerated form of agent. The behaviour of an object is simply to respond to every message sent to the object by executing the corresponding method. Based on this understanding of the relationship, our second design decision is to specify a multi-agent system as a set of agents, nothing but agents.

```
System ::= {Agent-description | class-description}*
```

The environment of an agent is a subset of the agents in the system that may influence its behaviour. The syntax for the description of environments is given below.

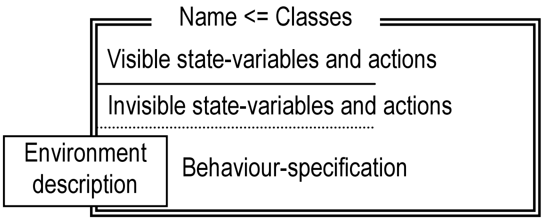
```
Environment-description ::= { name | All: class-name  | variable : class-name }*,
```

where a name indicates a specific agent in the system. 'All' means that all the agents of the class have influence on its behavior. A variable is a parameter in class specification. When instantiated, it indicates an agent in the class.

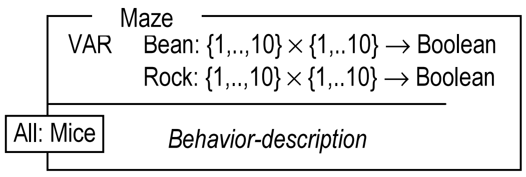
2.2 Classes of Agents as Birds of a Feather

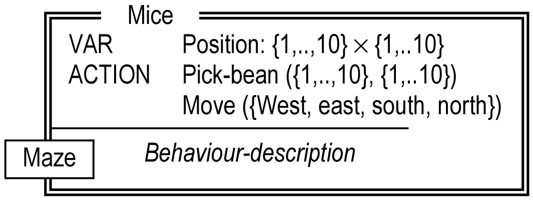
In object-oriented languages, a class is considered as the set of objects of common structure and function. Similarly, a class in SLAB is considered as a set of agents of same structural and behavioral characteristics. If an agent is specified as an instance of a class, it inherits the structure and behaviour descriptions from the class. However, in addition to those inherited structure and behaviour, an agent can also have additional behaviour and structure descriptions of its own. The syntax and graphic representation of class specification is given below.

```
class-description ::=
 class name [ <= {class-name} ]
     {instantiation};
     [ environment-description;]
     [ structure-description; ]
     [ behavior-description; ]
 end name
```


Name <= Classes
Visible state-variables and actions
Invisible state-variables and actions
Environment description
Behaviour-specification

For example, consider a system of mice in a maze. The maze consists of 10 by 10 squares. Each square can either be occupied by a rock or has a bean, or be empty. A mouse can move from one square to its adjacent square if the square is not occupied by a rock. It can pick up a bean if the square has a bean. The structure of the system can be specified by a class Mice and an agent Maze as below. The agent Maze represents the maze. It can be understood as the manager of the maze to up date the state of the maze when a mouse in the system picks up a bean. The specification of the dynamic behaviour will be given later.


Maze
VAR Bean: {1,..,10} × {1,..10} → Boolean
Rock: {1,..,10} × {1,..10} → Boolean
All: Mice
Behavior-description


Mice
VAR Position: {1,..,10} × {1,..10}
ACTION Pick-bean ({1,..,10}, {1,..10})
Move ({West, east, south, north})
Maze
Behaviour-description

As a template of agents, a class may have parameters. The variables specified in the form of “identifier: class-name” in the environment description are parameters. Such an identifier can be used as an agent name in the behaviour description of the class. When class name(s) are given in an agent specification, the agent is an instance of the classes. The instantiation clause gives the details about how the parameters are instantiated.

A class can also be defined as a subclass of existing classes by indicating the super-classes. A subclass inherits the structure and behaviour descriptions from its super-classes. It may also have some additional actions and obey some additional behaviour rules if they are specified in the subclass declaration. Some of the parameters of the super-class may also be instantiated in a subclass. As shown in section 3.2, the class and inheritance facilities provide a powerful vehicle to describe the normality of a society of agents. Multiple inheritances are allowed in the SLAB language to allow an agent to belong to more than one society and play more than one role in the system at the same time.

2.3 Scenarios as Patterns of Behaviours

The notion of scenario has been used in a number of areas in computing with different meanings. For example, in UML, scenarios are described as the sequences of messages passing between the system and the objects that represent the users. In the application of scenarios in testing software requirements [11], a scenario is described as an activity list that represents a task of human computer interaction. Generally speaking, a scenario is a set of situations that might occur in the operation of a system [12]. No matter how scenarios are described, its most fundamental characteristic is to put events in the context of the history of behaviour. Here, in a multi-agent system, we consider a scenario as a set of typical combinations of the behaviours of related agents in the system.

The use of scenarios and use cases in requirements analysis and specification has been an important part of object-oriented analysis, see for example, [13]. However, because an object must respond in a uniform way to all messages that call a method, there is a huge gap between scenarios and requirements models. The object-oriented paradigm is lack of a method to analyse the consistency between use cases (or scenarios) and requirements models and a method to synthesise requirements models from use cases or scenarios, although such methods exist for structured analysis [12]. As extensions to OO methodology, the use of scenarios in agent oriented analysis and design has been proposed by a number of researchers, for example [14, 15, 16]. In the design of SLAB, we recognised that scenarios can be more directly used to describe agent behaviour. The gap between scenarios and requirements models no longer exists in agent-based systems because the agent itself controls the its behaviour. Its responses can be different from scenario to scenario rather than be uniform to all messages that call a method.

In SLAB, a basic form of scenario description is a set of patterns. Each pattern describes the behaviour of an agent in the environment by a sequence of observable state changes and observable actions. A pattern is written in the form of [p_1, p_2, ..., p_n] where $n \geq 0$. Table 1 gives the meanings of the patterns.

```
pattern ::= [ { event || [ constraint ] } ]
event ::= [ time-stamp: ]  [ action ] [ ! state-assertion ]
action ::= atomic-pattern [ ^ arithmetic-expression ]
atomic-pattern ::= $  | ~ | action-variable | action-identifier [ ( { arithmetic-expression } ) ]
time-stamp ::= arithmetic-expression
```

where a constraint is a first order predicate.

Table 1. Meanings of the patterns

Pattern	Meaning
$	*wild card*, it matches with all actions
~	*silence* event
Action variable	It matches an action
P^k	a sequence of k events that match pattern P
Action $(a_1, a_2, ...a_k)$	*event* happens with parameters match $(a_1, a_2, ...a_k)$
$[p_1,..., p_n]$	The previous sequence of events match the patterns $p_1, ..., p_n$

In addition to the pattern of individual agents' behaviour, SLAB also provides facilities to describe global situations of the whole system. The syntax of scenarios is given below.

```
Scenario ::= Agent-identifier : pattern | arithmetic-relation
        | ∃ [arithmetic-exp] Agent-Var ∈ Class. Pattern | ∀Agent-Var ∈ Class. Pattern
        | scenario & scenario | scenario OR scenario | ~ scenario
```

where an arithmetic relation can contain an expression in the form of μAgent_var∈Class.Pattern, which is a function that returns the number of agents in the class whose behaviour matches the pattern. The semantics of the scenario descriptions are given in Table 2.

Table 2. Semantics of scenario descriptions

Scenario	Meaning
A: P	The situation when agent A's behavior matches pattern P
$\forall X \in C.P$	The situation when the behaviours of all agents in class C match pattern P
$\exists_{[m]} X \in C.P$	The situation when there exists at least m agents in class C whose behavior matches pattern P where the default value of the optional expression m is 1
$\mu X \in C.P=N$	The situation when N is the number of agents in class C whose behavior matches pattern P
S_1 & S_2	The situation when both scenario S_1 and scenario S_2 are true
S_1 OR S_2	The situation when either scenario S_1 or scenario S_2 or both are true
~ S	The situation when scenario S is not true

The following are some examples of scenario descriptions.

Example.

(1) Maze: $!\forall$n, m$\in$\{1,..,10\}. Bean(n, m)=False.

It describes the situation in the mice-maze system when there is no bean left in the maze.

(2) $\exists$ p$\in$ Parties. t_{2000}: [nominate-president(Bush)] || t_{2000}=(March/2000).

It describes the situation that at least one agent in the class called Parties took the action nominate-president(Bush) at the time of March 2000.

(3) (μ x$\in$ Citizen. [vote(Bush)] / μ x$\in$ Citizen. [$]) > 1/ 2

It describes the situation that more than half of the agents in the class Citizen took the action of vote(Bush).

2.4 Specification of Agent Behaviour

As discussed in section 2.1, an agent's autonomy is its capability of controlling its internal state and action. An agent changes its state and takes an action as a response to the situation in its environment rather than simply as a response to a request of its service. Various models of agents such as the BDI model have been proposed and investigated to represent and reason about agent's autonomous behaviour. The structure description facility that SLAB provides is intended to specify such structural model of agents. However, structural model alone is insufficient to specify agent's autonomous behaviour. We also need a facility to specify explicitly how the structural model (such as the belief, desire and intention) is related to actions and how observations of the environment are related to the changes at internal states. Among many possible forms of such a facility such as procedural specifications and temporal logic formulas, we believe that the most effective form is a set of transition rules.

Based on the characteristics of agent's behaviour, we recognised that a rule should contain the following parts:

- *Rule-name*: which enables us to indicate which rule is used in the reasoning of system's behaviour;
- *Scenario*: which specifies the situation when a rule is applicable;
- *Transition*: which specifies the action or state change to take place when the rule is applied;
- *Probability distribution*: the probability that the rule is applied when the scenario occurs;
- *Pre-condition*: the condition for the action to take place.

The syntax of a rule is given below.

Behaviour-rule ::= [<rule-name>] [prob:] pattern –> event, [Scenario] [*where* pre-cond] ;

In a behaviour rule, the pattern on the left-hand-side of the –> symbol describes the pattern of the agent's previous behaviour. The scenario describes the situation in the environment, which specifies the behaviours of the agents in its environment. The where-clause is the pre-condition of the action to be taken by the agent. The event on the right-hand-side of –> symbol is the action to be taken when the scenario happens and if the pre-condition is satisfied. The agent may have a none-deterministic behaviour. The expression prob in a behaviour rule is an expression that defines the probability for the agent to take the specified action on the scenario. When prob is the constant 1, it can be omitted. SLAB also allows specification of none-deterministic behaviour without giving the probability distribution. In such cases, '#' symbol is used to denote any probability that is greater than 0 and less than 1.

The following gives a behavior specification of the class Mice and the agent Maze. Micky is an instant of Mice. In addition to the inherited structure and behavior, it also remembers the number of beans it has picked up. It always first picks up a bean at the west or north adjacent square when there is a bean.

```
Mice
VAR    Position: {1,..,10} × {1,..10}
ACTION Pick-bean ({1,..,10}, {1,..10}), Move ({West, east, south, north})
-----------------------------------------------------------------
Maze    <Move west>  #: [!position =(n, m)] → Move(west) ! position =(n–1,m);
            if Maze: ! ~ Rock(n–1,m)
        <Move east>  #: [!position =(n, m)] → Move(east) ! position =(n+1,m);
            if Maze: ! ~ Rock(n+1,m)
        <Move south> #: [!position =(n, m)] → Move(south) ! position =(n,m+1);
            if Maze: ! ~ Rock(n,m+1)
        <Move north> #: [!position =(n, m)] → Move(north) ! position =(n,m–1);
            if Maze: ! ~ Rock(n,m–1)
        <Pick bean>: #: [!position =(n,m)]→Pick-bean(n,m); if Maze: !Bean(n,m)

Maze
VAR    Bean: {1,..,10} × {1,..10} → Boolean
       Rock: {1,..,10} × {1,..10} → Boolean
-----------------------------------------------------------------
All: Mice   <Update bean> [!Bean(n,m)]→!Bean(n,m)=False,
                if ∃x∈ Mice. (x:[Pick-bean(n, m)] )

Micky: Mice
-----------------------------------------------------------------
        VAR Beans: Integer
- - - - - - - - - - - - - - - - - - - - - - - - -
Maze    <Move west>  [!position =(n, m)] → Move(west) ! position =(n–1,m);
            if Maze: [!Bean(n–1,m)]
        <Move south> [!position =(n, m)] → Move(south) ! position =(n,m+1);
            if Maze: [!Bean(n,m+1)]
        <Pick bean>: #: [!position =(n, m)]→Pick-bean(n,m) ! Beans = Beans* + 1;
```

3 Examples

In this section, we give two examples to demonstrate SLAB's style and capability.

3.1 The Maxims System

Maes' Maxims system [17] is a personal assistant agent for handling emails. The system consists of Eudora and the user as the environment of the Maxims agent.

Eudora contains a number of mailboxes and can perform a number of operations on emails. The operations include reading a mail in the inbox, deleting a mail from the mail box, archiving a mail in a folder, sending a mail to a number of addresses, and forwarding a mail to a number of addresses. For the sake of simplicity, we assume that there is only one mailbox named as inbox in the following specification of the software. The behaviour of Eudora is a typical object's behaviour. That is, whoever sends a command to the Eudora, it will perform the corresponding operation. This behaviour is explicitly specified in the following SLAB specification through two facilities. Firstly, in the specification of its environment, it is made clear that all agents in the environment have influence on its behaviour. Secondly, in the specification of the behaviour rules, it is clearly specified that the only condition for Eudora to take an action is that some agent sends a command to ask it to do so.

```
Eudora
VAR         Inbox: list(Mail)          (* The inbox of the emails *)
ACTION      Read(Mail); Delete(Mail); Archive(Mail, Folder);
            Send(Mail, list(address)); Forward(Mail, list(address));
-----------------------------------------------------------
      <Read>      !mail∈ Inbox →Read(mail),
                        if ∃u:Agent. u:[Command(Eudora, Read(mail))];
All: Agent
      <Delete>    !mail∈ Inbox →Delete(mail) ! mail∉ Inbox,
                        if ∃u:Agent.u:[Command(Eudora, Delete(mail))];
      <Archive>   !mail∈ Inbox →Archive(mail, folder) !mail∉ Inbox & mail∈ folder,
                        if ∃u:Agent.u: [Command(Eudora, Archive(mail, folder))];
      <Send>      !mail∈ Inbox →Send(mail, list(address)),
                        if ∃u:Agent. u:[Command(Eudora, Send(mail, list(address))];
      <Forward>   !mail∈ Inbox →Forward(mail, list(address)),
                        if ∃u:Agent. u:[Command(Eudora, Forward(mail, list(address))];
```

A user's behavior is non-deterministic. The specification given below only shows the possible actions a user may take. There are two types of actions a user may take. One is to command an agent to take an action; the other is to grant an agent to take an suggested action. Notice that, the rules that specify the user's behaviour have an unknown probabilistic distribution.

```
User
ACTION   Command(Agent, Action)      (* Command the agent to take an action *)
         Grant(Agent, Action) (* Grant the agent to take a suggested action *)
-----------------------------------------------------------
<Grant suggestion> #: [$] → Grant(Maxim, action); if Maxim: [Suggest(Self, action)]
<Not grant suggestion> #: [$] → Command(Eudora, another-action);
Eudora       if Maxim: [Suggest(self, action)]  where (another-action ≠ action)
<Act-as-Predicted> #: [$] → action, if Maxim: [Predict(user, action)]
<Act-not-Predicted> #: [$] → action,
             if Maxim: [Predict(user, another-action)]  where (another-action ≠ action)
Maxim
<Set do-it level>    #: [$] → Command(Maxim, set-do-it(r)); where (0<r<1)
<Set tell-me level> #: [$] → Command(Maxim, set-tell-me(r)); where (0<r<1)
```

Maxims observes the user's actions and the state of Eudora. When a mail is delivered to Eudora's Inbox, Maxim finds out the best match in the set of emails that user has handled and the action that the user has taken in the situation. It, then, makes a suggestion or a prediction of user's actions. It also communicates with the user through facial expressions. Once the user grants a suggestion, Maxims commands Eudora to perform the action. These are specified by a set of rules. The <Command> rule states that maxims can command Eudora an operation on behalf of the user if its confidence level is greater than or equal to the do-it threshold[1]. The <Suggest> rule states that it makes a suggestion if the confidence level is higher than tell-me threshold but lower than do-it. The <Predict> rule states that it predicts the user's action if the confidence level is lower than tell-me. There are also rules in the specification of Maxims that specify its reaction to the user's responses to the agent's suggestions and predictions.

```
      Maxims
 VAR     Facial-expression: {Working, Suggestion, Unsure, Gratified, Pleased,
                             Surprised, Confused}
 Action  Command(Agent, Action); Suggest(Agent, Action); Predict(Agent, Action);
 ---------------------------------------------------------------------
 VAR     Tell-me-level, do-it-level : Real
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 <Suggest> [$] → Suggest(user, action) ! Facial-expression = Suggestion,
                    if Eudora: [!mail∈ Inbox] &  user: [X[n]^k],
 User               where (action, confidence) = Best-match(mail, {X[n]^k| n=1,..,k})
                       & tell-me-level ≤ confidence < do-it-level
 <Predict>   [$] → Predict(user, action) ! Facial-expression = Unsure,
                    if Eudora: [!mail∈ Inbox] &  user: [X[n]^k],
 Eudora             where (action, confidence) = Best-match(mail, {X[n]^k| n=1,..,k})
                       &  confidence < tell-me-level
 <Work>      [$] → action ! Facial-expression = Working,
                    if Eudora: [!mail∈ Inbox] &  user: [X[n]^k],
                    where (action, confidence) = Best-match(mail, {X[n]^k| n=1,..,k})
                       & do-it-level-level ≤  confidence
 <Set tell-me-level> [$] → ! tell-me-level = r, user: [Command(Self, set-tell-me(r))]
 <Set do-it-level>   [$] → ! do-it-level = r, user: [Command(Self, set-do-it(r))]
 <Gratified> [Suggest(user, action)] → action ! Facial-expression = Gratified,
                    if user:[Grant(self, action)];
 <Surprised> [Suggest(user, action)] → ! Facial-expression = Surprised,
                    if user:[another-action)], where another-action ≠ action
 <Pleased>  [Predict(user, action)] → ! Facial-expression = Pleased, if user:[action)];
 <Confused> [Predict(user, action)] → ! Facial-expression = Confused,
                    if user:[another-action)], where another-action ≠ action
 <Command> [Suggest(user, action] → action ! Facial-expression = Working,
                    if user:[Grant(self, action)]
```

[1] For the sake of space, the definition of the function *Best-match*: (*Mail* × List(*Mail* × *Action*)) → (*Action* × *Confidence-level*) is omitted.

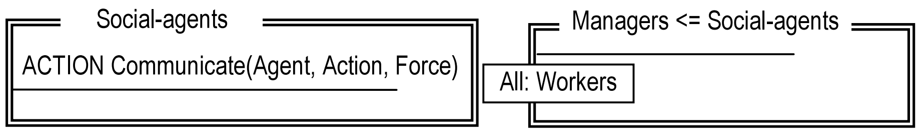
Maxims' autonomous behavior is reflected in the specification in SLAB. Firstly, it selectively observes the environment. It observes the state of Eudora to determine if there is a mail in its Inbox. It also observes the action taken by the user to learn from the user's behaviour. Secondly, as discussed above, its behaviour is not simply determined by the event, but also the history of the user's behaviour. It can even take actions without the user's command. Of course, an agent may also have a part of behaviour that simply obeys the user's command. The maxims agent obeys the user's commands on setting tell-me and do-it thresholds. The rules <Set do-it-level> and <Set tell-me-level> specify such behaviour.

3.2 Speech Act and Collaborative Behaviour

In a multi-agent system, agents communicate to each other and collaborate with each other. To illustrate SLAB's capability of specification of such behaviour, we describe the differences between illocutionary forces in communications as their effects on agent behaviour. As in [18, 19], illocutionary forces are classified into 7 types.

Force = {Assertive, Directive, Commissive, Permissive, Prohibitive, Declarative, Expressive}

We first define a class of agents, called social-agents, which can communicate with each other. Informally, the action Communicate(X, Y, Z) is to send a message Y to agent X with illocutionary force Z, where the message Y is an action. The meaning of the communication depends on the illocutionary force, which is understood by all the agents in the society. An approach to the formal specification of the meaning of an illocutionary force is to define a set of rules for how agents should interpret such communications. However, in a human society, people play different roles. The same sentence may have different effect depending on who says to whom. For example, a commander can give an order to a soldier and expect the soldier to perform the action as ordered. However, if the same message communicates in the opposite direction from the soldier to the commander, it is not socially acceptable and one would not expect an action to be taken.


Social-agents
ACTION Communicate(Agent, Action, Force)
Managers <= Social-agents
All: Workers

Therefore, instead of giving a set of behaviour rules for all agents to interpret the meanings of illocutionary forces in the same way, we add an additional twist to show how to specify the situation in which different agents can interpret differently according their roles in the system. The situation specified in the example is at work place, where the agents are divided into two groups: the workers who perform various tasks and the managers who assign tasks to workers.

There are two basic requirements of a worker agent. One is to follow the orders of its manager; and the other is to report to his manager when it finishes a task. The <Obey-Order> rule in the class Workers specifies that a worker agent must take the order from its manager agent Boss (which is a parameter of the class) and perform the action as the boss ordered. The <Report> rule specifies that after a worker agent finishes a job, it must report to its boss. Every worker agent must satisfy these rules, but a manager agent does not need to. Here, we simplified the situation that managers themselves are not organised in a hierarchical structure.

```
Workers <= Social-agents
--------------------------------------------------------------------
Boss: Managers
--------------------------------------------------------------------
<Obey-Order> [$] → action, if Boss:[Communication(X, action, directive)]
    where (X = Self or X = All)
<Report> [action ! state] → Communication(Boss, action ! state, declarative);
    where (action ≠ Communication(Boss, x, y))
```

The personality of an agent can also be described by rules governing its behavior. For example, a naughty apprentice would take any chance to play (i.e. take actions that are neither ordered nor approved by the boss, even prohibited by the boss, when the boss is busy in communication with other workers and does not give him an order for a while. However, as a worker, an apprentice will still obey the orders of the boss. Such a behavior is specified in the class Apprentices. In contrast, an experienced worker will not only follow the rules but also do more than what is ordered.

```
Apprentices <= Workers
--------------------------------------------------------------------
Boss: Managers
--------------------------------------------------------------------
<Free time>  [$] → action; if Boss: [Communication(Xn, Mn, Fn)^5]n={1..5},
             where ∀n∈{1..5}. (Xn≠Self & Xn≠All & Fn≠Directive)
```

```
Experienced <= Workers
--------------------------------------------------------------------
Boss: Managers
--------------------------------------------------------------------
<Active>  [$] → Communication(Boss, action, commisive);
          if Boss: [Communication(X, action, expressive)],
          where (X = Self  or X = All)
<Keep promises> [Communication(X, action, commisive), $^k ]
          → action; where k<∞;
<Follows rules>  [$] → anyaction;
          if Boss:[Communication(X, action, prohibitive), $^k]
          where anyaction ≠ action & (X= Self or X= All)
          &   $ ≠ Communication(X, action, permissive)
```

4 Discussion

The design of the specification language SLAB is being investigated in the context of software engineering towards a methodology of agent-oriented analysis and design. In this paper, we argued that agents are encapsulations of behaviors. They are an extension of objects rather than a specialization of objects as in AOP [20]. We believe that the power of an agent-oriented approach comes from the encapsulation of behaviors in computational entities whose dynamic behavior can be determined by design objectives, not simply controlled by the environment. The key difference between an agent and an object is that the former has autonomous behavior while the later do not. We define autonomous behavior as the capability of deciding whether and when to take an action and which action to take. A special case is to take an action whenever a message is received and to perform the action as the message requested. This is how an object behaves. An agent's behavior can be more complicated than an object can because it can make decisions according to the scenario in the environment and its internal state. As many approaches proposed in the literature, the SLAB language provides facilities to specify what are the agents (which include objects) in a system, how an agent behaves, what is the environment of an agent, and what is the structure of its internal state as well. Moreover, SLAB further encapsulates these aspects in one entity for each agent.

Treating agents as the basic entity of a system enables us to describe situations in the operation of a system in terms of the behaviors of the agents and to formally specify such situations as scenarios. In this paper, we introduced a formalism to describe various types of scenarios including that when a system's agents are behaving in certain patterns as individuals, the existence of a number of agents behaving in a certain pattern, and the uniformity of the behaviors of the agents in the system, and statistical situations such as a specific proportion of agents behaving in a certain pattern. This formalism is at a very high abstraction level that describes what is the designed behavior rather than how such behavior is achieved. The examples presented in the paper demonstrated its power of specifying the behavior of an agent or a class of agents by a set of rules governing their reactions in various scenarios. Moreover, it also enables us to specify the behavior of a system in terms of a set of desirable scenarios.

A class in SLAB contains all the agents declared to be of the same structural and behavioral characteristics. The example given in the paper shown that social normality can be specified as a class of agents that obeys a common set of behavior rules. The inheritance mechanism enables software engineers to specify subclasses of agents that can have more specific behaviors. For example, the diversity in behavior due to differences in an agent's personality and the role an agent plays in a multi-agent system can be described by inheritance relations plus a set of additional behavior rules. A question need further investigation is whether redefinition of a rule should be allowed. The advantages are obviously the flexibility and expressiveness, but the disadvantages are also obvious which include the complexity due to none-monotonic natural of redefinition.

The work of this paper is a part of the research in progress. We are further investigating the following issues related to the SLAB language.

The SLAB language is developed on the base of STOC specification language [21] in which objects are active and persistent. It can describe timing and probabilistic features of real-time systems using a set of rules similar to that in SLAB. However, SLAB extended behavior patterns into scenarios and introduced the notion of visible and invisible states and actions and allowed states to be higher order entities. The semantic model of STOC is stochastic process. The consistence and completeness of a specification can be defined as the existences and uniqueness of a stochastic process model that satisfies the rules. For a large subset of formal specifications in STOC, effective algorithms exist to check the consistence and completeness. We are further studying if a similar semantic model can be used to define the semantics of SLAB and the algorithms for checking consistence and completeness.

A major challenge is to define a type system for the language. In this paper, we have treat types in a very intuitively, such as the type Agent represents all agents in the system, Action represents all actions an agent can take, and so on. It is recognized that many entities in SLAB specification, including Agents and Action, are not mathematical objects of a fix order. For example, consider a situation that one day a school boy Simon had a special event after school, hence he would stay at school a little longer than usual. He called mum Mary telling her that he was to be picked up at 4 o'clock at school. Mary then called the child minder Cheril to change the pick up time and then called her husband Howard telling him the story so that he would not expect Simon to be home until 4 o'clock. The following scenario in SLAB describes the situation.

```
Simon: [ t0: Communicate(Mary, Pick-up(Simon, 4pm, school), Directive)] &
Mary:  [ t1: Communicate(Cheril, Pick-up(Simon, 4pm, school), Directive),
         t2: Communicate(Howard, Communicate(Cheril, Pick-up(Simon, 4pm, school),
             Directive), Declarative)] || t1>t0
```

Notice that, mathematically speaking the actions in the scenario are of different orders. This gives the difficulty to define the type of Communicate.

By applying the theory of stochastic process, we can analysis and prove two types of properties of a system. Firstly, we can prove the probability of the occurrence of a certain scenario. Secondly, we can analyse if the stochastic process will converge to a stable state when the system executes long enough, and what is the stable state. Both types of properties are very important in the analysis of agent-based systems, but they are not supported by existing formal methods. We are investigating a formal method based on the theory of probability and stochastic process that support such analysis and proof using formal specification in SLAB. Moreover, given a SLAB specification of an agent-based system, the correctness of an implementation can be verified by proving that each agent's behavior satisfies the behavior rules. We are also investigating a calculus that supports proofs of behavioral equivalence between two agents. Moreover, the rules can also be naturally developed into test cases for the validation of the system. The scenario part in a rule forms a description of the test input, and the event part describes the expected action and output of the agent.

References

1. Rao, A. S., Georgreff, M. P.: Modeling Rational Agents within A BDI-Architecture. in Proc. of the International Conference on Principles of Knowledge Representation and Reasoning (1991) 473–484. Also in [9], 317–328.
2. Wooldridge, M., Jennings, N. R.: Formalizing The Cooperative Problem Solving Process. in Proc. of 13th International Workshop on Distributed Artificial Intelligence (1994) 403–417. Also in [9] 430–440.
3. Myer, J-J., Schobbens, P-Y. (eds.): Formal Models of Agents - ESPRIT Project ModelAge Final Workshop Selected Papers. LNAI 1760, Springer, Berlin Heidelberg (1999)
4. Chainbi, W., Jmaiel, M., Abdelmajid, B. H.: Conception, Behavioural Semantics and Formal Specification of Multi-Agent Systems. in Zhang, C., Lukose, D. (eds): Multi-Agent Systems: Theories, Languages, And Applications, 4th Australian Workshop on Distributed Artificial Intelligence Selected Papers. Bristane, QLD, Australia, July 1998. LNAI 1544. Springer, Berlin Heidelberg New York (1998) 16–28.

5. Brazier, F. M. T., Dunin-Keplicz, B. M., Jennings, N. R., Treur, J.: DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. in Int. Journal of Cooperative Information Systems. **1**(6) (1997) 67–94.
6. Conrad, S., Saake, G., Turker, C.: Towards an Agent-Oriented Framework for Specification of Information Systems. in [3] (1999) 57–73.
7. Jennings, N. R.: Agent-Oriented Software Engineering. in Garijo, F. J., Boman, M. (eds.): Multi-Agent System Engineering, Proceedings of 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Valencia, Spain, June/July 1999. LNAI 1647. Springer, Berlin Heidelberg New York (1999) 1–7.
8. Spivey, J. M.: The Z Notation: A Reference Manual. (2nd edition). Prentice Hall (1992).
9. Huhns, M., Singh, M. P. (eds.): Readings in Agents. Morgan Kaufmann, San Francisco (1997)
10. Jennings, N. R., Wooldridge, M. J. (eds.): Agent Technology: Foundations, Applications, And Markets. Springer, Berlin Heidelberg New York (1998)
11. Zhu, H., Jin, L., Diaper, D.: Application of Task Analysis to the Validation of Software Requirements, Proc. SEKE'99. Kaiserslautern, Germany, (June, 1999) 239–245.
12. Zhu, H., Jin, L.: Scenario Analysis in An Automated Requirements Analysis Tool. Technical Report, CMS–TR–00–01, School of Computing and Mathematical Sciences, Oxford Brookes University, (Jan. 2000). in Requirements Engineering Journal (*in press*)
13. Jacobson, I., *et al.*: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992)
14. Iglesias, C. A., Garijo, M., Gonzalez, J. C., Velasco, J. R.: Analysis And Design of Multiagent Systems Using MAS-Common KADS. in Singh, M. P., Rao, A., Wooldridge, M. J. (eds.): Intelligent Agents IV. LNAI 1356. Springer, Berlin Heidelberg New York (1998) 313–327.
15. Iglesias, C. A., Garijo, M. Gonzalez, J. C.: A Survey of Agent-Oriented Methodologies. in Muller, J. P., Singh, M. P., Rao, A., (eds.): Intelligent Agents V. LNAI 1555. Springer, Berlin Heidelberg New York (1999) 317–330.
16. Moulin, B. Brassard, M.: A Scenario-Based Design Method And Environment for Developing Multi-Agent Systems. in Lukose, D., Zhang, C. (eds.): Proc. of First Australian Workshop on DAI. LNAI 1087. Springer Verlag, Berlin Heidelberg New York (1996) 216–231.
17. Maes, P.: Agents That Reduce Work And Information Overload, Communications of the ACM, **37**(7) (1994) 31–40.
18. Singh, M. P.: A Semantics for Speech Acts. Annals of Mathematics and Artificial Intelligence. **8**(I–II) (1993) 47–71.
19. Singh, M. P.: Agent Communication Languages: Rethinking the Principles. IEEE Computer (Dec. 1998) 40–47.
20. Shoham, Y.: Agent-Oriented Programming. Artificial Intelligence. **60**(7) (1993) 51–92.
21. Zhu, H., Jin, L.: A Specification Language of Stochastic Real-Time Systems. in Proc. SEKE'97. Madrid, Spain (June 1997) 358–365.

APT Agents: Agents That Are Adaptive, Predictable, and Timely

Diana F. Gordon

AI Center, Naval Research Laboratory, Washington D.C. 20375
gordon@aic.nrl.navy.mil,
http://www.aic.nrl.navy.mil/~gordon

Abstract. The increased prevalence of agents raises numerous practical considerations. This paper addresses three of these – adaptability to unforeseen conditions, behavioral assurance, and timeliness of agent responses. Although these requirements appear contradictory, this paper introduces a paradigm in which all three are simultaneously satisfied. Agent strategies are initially verified. Then they are adapted by learning and formally reverified for behavioral assurance. This paper focuses on improving the time efficiency of reverification after learning. A priori proofs are presented that certain learning operators are guaranteed to preserve important classes of properties. In this case, efficiency is maximal because no reverification is needed. For those learning operators with negative a priori results, we present incremental algorithms that can substantially improve the efficiency of reverification.

1 Introduction

Agents (e.g., robots or softbots) are becoming an increasingly prevalent paradigm. Many systems of the future will be multiagent. The agents paradigm offers numerous advantages, such as flexibility and fault-tolerance. However it also introduces new challenges.

Consider an example. The forthcoming field of nanomedicine holds great promise for microsurgery. Medical nanorobots (also called "nanobot" agents), with tiny sensors and medical devices, will be capable of performing delicate, fine-grained operations within the human body. This would revolutionalize the field of medicine. It requires precise navigation through the body, and sophisticated multiagent positioning. For example, multiple nanobots could form a flexible surface comprised of independently controllable agents that translate or rotate their positions relative to each other [8]. Because each human body is unique, these nanobots need to adapt their surgical strategy to the individual.

Unfortunately, by providing agents the capability to adapt, we risk introducing undesirable behavioral side-effects – particularly in situations where global system behavior may be significantly affected by a minor local change. How can we guarantee that agents will achieve desirable global coordination? For example, we want assurance that the actions of nanobots performing tissue repair will not conflict with the actions of nanobots performing cancer cell removal.

Formal verification can be applied to ensure proper global multiagent coordination. Unfortunately, though, verification can be quite slow. This raises the issue of timeliness. Using the medical nanobots example, agents performing tissue repair could provide formal guarantees that their actions will not conflict with those of other nanobots; yet if the process of verification takes too long, the patient might suffer harm.

In response to problems such as these, we have developed *APT agents*, i.e., agents that are simultaneously adaptive, predictable and timely. Adaptation is achieved with machine learning/evolutionary algorithms, predictability with formal verification, and timeliness by exploiting the knowledge that learning has occurred to streamline the formal verification. To achieve APT agents, we have developed efficient methods for determining whether the behavior of adaptive agents remains within the bounds of pre-specified constraints (called "properties") after learning. This includes verifying that properties are preserved for single adaptive agents as well as verifying that global properties are preserved for multiagent systems in which one or more agents may adapt.

There has been a growing body of research on learning/adaptive agents (e.g., [14]), as well as evolving agent strategies (e.g., [7]). However, that research does not address formally verifying agents' behavior following adaptation. Our approach includes both adaptation and formal verification – the latter using model checking. Model checkers can determine whether an agent plan (strategy) S satisfies a property P, i.e., $S \models P$. Although highly effective, model checking has a time complexity problem. Suppose that in a multiagent system, each agent has its own plan. To verify a global property P, the (synchronous) product has to be taken of the individual agent plans to form a multiagent plan S, which is then verified. Model checking global properties of a multiagent plan has time complexity that is exponential in the number of agents. With a large number of agents, this is seriously problematic. In fact, even model checking a single agent plan with a huge number of states can be computationally prohibitive. A great deal of research in the verification community is currently focused on developing reduction techniques for handling very large state spaces [5]. Nevertheless, none of these techniques are tailored specifically for *efficient re*-verification after learning has altered the system (which is the focus of this paper). There are a few methods in the literature that are designed for software that changes. One that emphasizes efficiency, as ours does, is [16]. However none of them (including [16]) are applicable to multiagent systems in which a single agent could adapt, thereby altering the global behavior of the overall system. In contrast, our approach addresses the timeliness of adaptive multiagent systems.

In our APT agents framework (see Figure 1), there are one or more agents with "anytime" plans, i.e., plans that are continually executed in response to internal and external environmental conditions. Each agent's plan is assumed to be in the form of a finite-state automaton (FSA). FSAs have been shown to be effective representations of reactive agent plans/strategies (e.g., [4], [7], [11]).

Let us begin with step (1) in Figure 1. There are at least a couple of ways that the FSA plans could be formed initially. For one, a human plan designer could

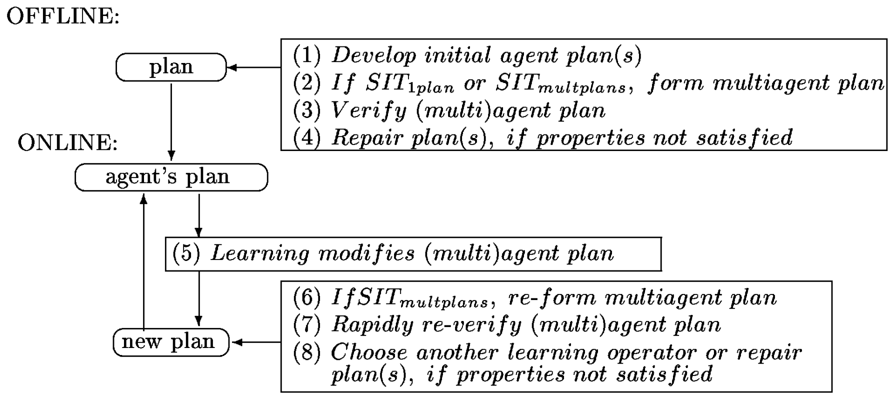

OFFLINE:
plan
(1) Develop initial agent plan(s)
(2) If SIT_{1plan} or $SIT_{multplans}$, form multiagent plan
(3) Verify (multi)agent plan
(4) Repair plan(s), if properties not satisfied
ONLINE:
agent's plan
(5) Learning modifies (multi)agent plan
new plan
(6) If $SIT_{multplans}$, re-form multiagent plan
(7) Rapidly re-verify (multi)agent plan
(8) Choose another learning operator or repair plan(s), if properties not satisfied

Fig. 1. APT agents framework.

engineer the initial plans. This may require considerable effort and knowledge. An appealing alternative is to evolve (i.e., learn using evolutionary algorithms) the initial plans in a simulated environment [7].

Human plan engineers or evolutionary algorithms can develop plans that satisfy agents' goals to a high degree, but to provide strict behavioral (especially global) guarantees, formal verification is also required. Therefore, we assume that prior to fielding the agents, the (multi)agent plan has been verified offline to determine whether it satisfies critical properties (steps (2) and (3)). These properties can either be expressed in linear temporal logic, or in the form of automata if automata-theoretic (AT) model checking [6] is done. If a property fails to be satisfied, the plan is repaired (step (4)). Steps (2) through (4) require some clarification. If there is a single agent, then it has one FSA plan and that is all that is verified and repaired, if needed. We call this SIT_{1agent}. If there are multiple agents that cooperate, we consider two possibilities. In SIT_{1plan}, every agent uses the same multiagent plan that is the product of the individual agent plans. This multiagent plan is formed and verified to see if it satisfies global multiagent coordination properties. The multiagent plan is repaired if verification produces any errors, i.e., failure of the plan to satisfy a property. In $SIT_{multplans}$, each agent independently uses its own individual plan. To verify global properties, one of the agents takes the product of these individual plans to form a multiagent plan. This multiagent plan is what is verified. For $SIT_{multplans}$, one or more individual plans is repaired if the property is not satisfied.

After the initial plan(s) have been verified and repaired, the agents are fielded. While fielded (online), the agents apply learning to their plan(s) as needed (step (5)). Learning (e.g., with evolutionary operators) may be required to adapt the plan to handle unexpected situations or to fine-tune the plan. If SIT_{1agent} or SIT_{1plan}, the single (multi)agent plan is adapted. If $SIT_{multplans}$, each agent adapts its own FSA, after which the product is formed. For all situations, one agent then rapidly *re*-verifies the new (multi)agent plan to ensure it still satisfies the required properties (steps (6) and (7)). Whenever (re)verification fails, it produces a counterexample that is used to guide the choice of an alternative learning operator or other plan repair as needed (step (8)). This process of executing, adapting, and reverifying plans cycles indefinitely as needed. The main focus of this paper is steps (6) and (7). The novelty of the approach outlined in our framework is not in machine learning or verification per se, but rather the combination of the two.

Rapid reverification after learning is a key to achieving timely agent responses. Our results include proofs that certain useful learning operators are *a priori* guaranteed to be "safe" with respect to important classes of properties, i.e., if the property holds for the plan prior to learning, then it is guaranteed to still hold after learning. If an agent uses these "safe" learning operators, it will be guaranteed to preserve the properties with *no re*-verification required, i.e., steps (6) through (8) in Figure 1 need not be executed. This is the best one could hope for in an online situation where rapid response time is critical. For other learning operators and property classes our a priori results are negative. However, for these cases we have novel *incremental* reverification algorithms that can save time over total reverification from scratch.

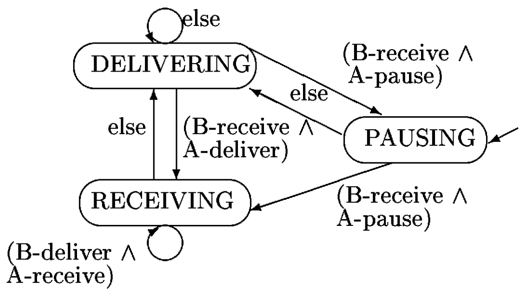


Fig. 2. A finite-state automaton plan for agent A.

2 Agent Plans

Figure 2 shows a finite-state automaton (FSA) plan for an agent that is, perhaps, a nanobot's protocol for exchanging tissue with another nanobot. The agent may be in a finite number of states, and actions enable it to transition from state to state. Action begins in an initial (i.e., marked by an incoming arrow from nowhere) state (PAUSING in Figure 2). The *transition conditions* (i.e., the Boolean algebra expressions labeling the edges) in an FSA plan succinctly describe the set of actions that enable a state-to-state transition to occur. The operator $\wedge$ means "AND," $\vee$ means "OR," and $\neg$ means "NOT." We use $V(S)$, $I(S)$, $E(S)$, and $M(v_i, v_j)$ to denote the sets of vertices (states), initial states, edges, and the transition condition from state v_i to state v_j for FSA S, respectively. $\mathcal{L}(S)$ denotes the language of S, i.e., the set of all action sequences that begin in an initial state and satisfy S's transition conditions. These are the action sequences (called *strings*) allowed by the plan.

Note that the transition conditions of one agent can refer to the actions of one or more other agents. This allows each agent to be reactive to what it has observed other agents doing. Nevertheless, the agents do not have to synchronize on their choice of actions. The only synchronization is the following. At each time step, all of the agents independently choose an action to take. Then they observe the actions of other agents that are referenced in their plan (e.g., agent A's transition conditions mention agent B's actions; therefore A needs to observe what its neighbor, B, did). Based on its own action and those of the other referenced agent(s), the agent knows the next state to which it will transition. The FSAs are assumed to be deterministic and complete, i.e., for every allowable action there is a unique next state.

In SIT_{1plan} or $SIT_{multplans}$, there are multiple agents. Prior to initial verification (and in $SIT_{multplans}$ this is also needed prior to subsequent verification), the synchronous multiagent product FSA is formed. This is done in the standard manner, by taking the Cartesian product of the states and the intersection of the transition conditions. Model checking then consists of verifying that all sequences of multiagent actions allowed by the product plan satisfy the property.

Each multiagent action is an *atom* of the Boolean algebra used in the product FSA transition conditions. To help in understanding the discussions below, we briefly define a Boolean algebra atom. In a Boolean algebra $\mathcal{K}$, there is a partial order among the elements, $\preceq$, which is defined as $x \preceq y$ if and only if $x \wedge y = x$. (It may help to think of $\preceq$ as analogous to $\subseteq$ for sets.) The distinguished elements 0 and 1 are defined as $\forall x \in \mathcal{K}$, $0 \preceq x$ and $\forall x \in \mathcal{K}$, $x \preceq 1$. The atoms (analogous to single-element sets) of $\mathcal{K}$, $\Gamma(\mathcal{K})$, are the nonzero elements of $\mathcal{K}$ minimal with respect to $\preceq$. For example, (B-receive $\wedge$ A-pause) is an atom, or multiagent action, when there are two agents (A and B). If the agents take multiagent action x, then each agent will transition from its state v_1 to state v_2 whenever $x \preceq M(v_1, v_2)$.

Now we can formalize $\mathcal{L}(S)$. A string (action sequence) $\mathbf{x}$ is an infinite-dimensional vector, $(x_0, ...) \in \Gamma(\mathcal{K})^\omega$. A *run* $\mathbf{v}$ *of string* $\mathbf{x}$ is a sequence $(v_0, ...)$ of vertices such that $\forall i$, $x_i \preceq M(v_i, v_{i+1})$. In other words, a run of a string is the sequence of vertices visited in an FSA when the string satisfies the transition conditions along the edges. Then $\mathcal{L}(S) = \{\mathbf{x} \in \Gamma(\mathcal{K})^\omega \mid \mathbf{x}$ has a run $\mathbf{v} = (v_0, ...)$ in S with $v_0 \in I(S)$ $\}$. Such a run is called an *accepting run*, and S is said to *accept* string $\mathbf{x}$.

3 Adaptive Agents: The Learning Operators

First, let us define the type of machine learning performed by the agents. A machine learning operator $o : S \to S'$ changes a (product or individual) FSA S to post-learning FSA S'. For a complete taxonomy of our learning operators, see [9]. Here we do not address learning that adds/deletes/changes states, nor do we address learning that alters the Boolean algebra used in the transition conditions, e.g., via abstraction. Results on abstraction may be found in [10]. Instead, we focus here on edge operators.

Let us begin with our most general learning operator, which we call o_{change}. It is defined as follows. Suppose $z \preceq M(v_1, v_2)$, $z \neq 0$, for $(v_1, v_2) \in E(S)$ and $z \not\preceq M(v_1, v_3)$ for $(v_1, v_3) \in E(S)$. Then $o_{change}(M(v_1, v_2)) = M(v_1, v_2) \wedge \neg z$ (step 1) and/or $o_{change}(M(v_1, v_3)) = M(v_1, v_3) \vee z$ (step 2). In other words, o_{change} may consist of two steps – the first to remove condition z from edge (v_1, v_2) and the second to add condition z to edge (v_1, v_3). Alternatively, o_{change} may consists of only one of these two steps. Sometimes (e.g., in Subsection 5.3), for simplicity we assume that z is a single atom, in which case o_{change} simply changes the next state after taking action z from v_2 to v_3. A particular *instance* of this (or any) operator is the result of choosing v_1, v_2, v_3 and z.

Four one-step operators that are special cases of o_{change} are: o_{add} and o_{delete} to add and delete FSA edges (if a transition condition becomes 0, the edge is considered to be deleted), and o_{gen} and o_{spec} to generalize and specialize the transition conditions along an edge. Generalization adds actions to a transition condition (with $\vee$), whereas specialization removes actions from a transition condition (with $\wedge$). An example of generalization is the change of the transition condition (B-deliver $\wedge$ A-receive) to ((B-deliver $\wedge$ A-receive) $\vee$ (B-receive $\wedge$ A-receive)). An example of specialization would be changing the transition condition (B-deliver) to (B-deliver $\wedge$ A-receive).

Two-step operators that are special cases of o_{change} are: $o_{delete+gen}$, $o_{spec+gen}$, $o_{delete+add}$, and $o_{spec+add}$. These operators move an edge or part of a transition condition from one outgoing edge of vertex v_1 to another outgoing edge of vertex v_1. An example, using Figure 2, might be to delete the edge (RECEIVING, RECEIVING) and add its transition condition via generalization to (RECEIVING, DELIVERING). Then the latter edge transition condition would become 1. The two-step operators preserve FSA determinism and completeness. One-step operators must be paired with another operator to preserve these constraints.

When our operators are used in the context of evolving FSAs, each operator application is considered to be a "mutation." For applicability of the incremental reverification algorithms in Subsection 5.2, we assume that learning is incremental, i.e., at most one operator is applied to one agent per time step (or per generation if using evolutionary algorithms – this is a reasonable mutation rate for these algorithms [1]). Gordon [9] defines how each of the learning operators translates from a single agent FSA to a multiagent product FSA. The only translation we need to be concerned with here is that a single agent o_{gen} may translate to a multiagent o_{add}. We will see the implications of this in Subsections 5.2 and 5.3.

To understand some of the results in Subsection 5.1, it is necessary to first understand the effect that learning operators can have on accessibility. We begin with two basic definitions of accessibility:

Definition 1. *Vertex v_n is accessible from vertex v_0 if and only if $\exists$ a path (i.e., a sequence of edges) from v_0 to v_n.*

Definition 2. *Atom (action) $a_{n-1} \in \Gamma(\mathcal{K})$ is accessible from vertex v_0 if and only if $\exists$ a path from v_0 to v_n and $a_{n-1} \preceq M(v_{n-1}, v_n)$.*

Accessibility from initial states is central to model checking, and therefore changes in accessibility introduced by learning should be considered. There are two fundamental ways that our learning operators may affect accessibility: *locally* (abbreviated "L"), i.e., by directly altering the accessibility of atoms or states; *globally* (abbreviated "G"), i.e., by altering the accessibility of any states or atoms that are not part of the learning operator definition. For example, any change in accessibility of v_1, v_2, v_3, or atoms in $M(v_1, v_2)$ or $M(v_1, v_3)$ in the definition of o_{change} is considered local; other changes are global.

The symbol $\uparrow$ denotes "can increase" accessibility, and $\not\uparrow$ denotes "cannot increase" accessibility. We use these symbols with G and L, e.g., $\uparrow$ G means that a learning operator can (but does not necessarily) increase global accessibility. The following results are useful for Section 5:

– o_{delete}, o_{spec}, $o_{delete+gen}$, $o_{spec+gen}$: $\not\uparrow$ G $\not\uparrow$ L
– o_{add}: $\uparrow$ G $\uparrow$ L
– o_{gen}: $\not\uparrow$ G $\uparrow$ L
– $o_{delete+add}$, $o_{spec+add}$, o_{change}: $\uparrow$ G

Finally, consider a different characterization (partition) of the learning operators, which is necessary for understanding some of the results in Subsection 5.1. For this partition, we distinguish those operators that can introduce at least one new string (action sequence) with an infinitely repeating substring (e.g., (a,b,c,d,e,d,e,d,e,...) where the ellipsis represents infinite repetition of d followed by e) into the FSA language versus those that cannot. The only operators belonging to the second ("cannot") class are o_{delete} and o_{spec}.

4 Predictable Agents: Formal Verification

Recall that in SIT_{1plan} or $SIT_{multplans}$, there are multiple agents, and prior to initial verification the product FSA is formed (step (2) of Figure 1). In *all three* situations, to perform automata-theoretic (AT) model checking, the product must be taken with the FSA of $\neg P$ for property P (see below). We call the algorithm for forming the product FSA $Total_{prod}$.

Our general verification algorithm (not tailored to learning) is AT model checking. In AT model checking, the negation of the property is expressed as an FSA. Asking whether $S \models P$ is equivalent to asking whether $\mathcal{L}(S) \subseteq \mathcal{L}(P)$ for property P. This is equivalent to $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$, which is algorithmically tested by first taking the product ($\otimes$) of the plan FSA S and the FSA corresponding to $\neg P$, i.e., $S \otimes \neg P$. The FSA corresponding to $\neg P$ accepts $\overline{\mathcal{L}(P)}$. The product implements language intersection. The algorithm then determines whether $\mathcal{L}(S \otimes \neg P) \neq \emptyset$, which implies $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} \neq \emptyset$ ($S \not\models P$). This determination is implemented as a check for undesirable cycles in the product FSA $S \otimes \neg P$ that are accessible from some initial state.

The particular AT verification algorithm we have chosen is a simple, elegant one from Courcoubetis et al. [6]. This algorithm, which we call $Total_{AT}$, is designed to do a depth-first search through the FSA starting at initial states and visiting all states reachable from the initial states – in order to look for all verification errors, i.e., failures to satisfy the property. The algorithm assumes properties are represented as Büchi FSAs [3]. [1]

This paper focuses mainly on Invariance and Response properties – because these are considered to be particularly relevant for multiagent systems. Invariance properties ($\Box\neg p$, i.e., "always not p") can be used to prohibit deleterious multiagent interactions. For example $\Box\neg$ (B-deliver $\wedge$ A-deliver) states that agent B cannot take action B-deliver at the same time that agent A is taking action A-deliver. Response properties ($\Box(p \rightarrow \Diamond q)$ i.e., "always if trigger p occurs then eventually response q will occur") can ensure multiagent coordination by specifying that one agent's actions follow those of another in a particular sequence. For example $\Box$ (C-deliver $\rightarrow \Diamond$ A-receive) states that whenever agent C delivers something A must eventually receive it. Although C is not mentioned in A's plan, this property can be verified for the three-agent (A, B, C) product FSA.

5 APT Agents: Time-Efficient Reverification

Total reverification is time-consuming. For the sake of timely agents, we first tried to find as many positive a priori results as possible for our operators. Recall that a positive a priori result implies *no* reverification is required.

5.1 A Priori Results

Our objective is to lower the time complexity of reverification. The ideal solution is to identify *"safe" machine learning operators* (SMLOs), i.e., machine learning operators that are a priori guaranteed to preserve properties and require no run-time cost. For a plan S and property P, suppose verification has succeeded prior to learning, i.e., $S \models P$. Then a machine learning operator $o(S)$ is an SMLO if and only if verification is guaranteed to succeed after learning, i.e., if $S' = o(S)$, then $S \models P$ implies $S' \models P$.

We next present theoretical results. Proofs for all theorems may be found in [9].[2] Our two initial theorems, Theorems 1 and 2, which are designed to address the one-step operators, may not be immediately intuitive. For example, it seems reasonable to suspect that if an edge is deleted somewhere along the path from

[1] Because a true Response property cannot be expressed as a Büchi FSA, we use a First-Response property approximation. This suffices for our experiments [9].

[2] Properties are assumed to be expressed in linear temporal logic or as FSAs.

a trigger to a response, then this could cause failure of a Response property to hold – because the response is no longer accessible. In fact, this is not true. What actually happens is that deletions reduce the number of strings in the language. If the original language satisfies the property then so does is the smaller language. Theorem 1 formalizes this.

Theorem 1. *Let S and S' be FSAs, where S' is identical to S, but with additional edges. We define $o : S \rightarrow S'$ as $o : E(S) \rightarrow E(S')$, where $E(S) \subseteq E(S')$. Then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$.*

Corollary 1. *o_{delete} is an SMLO with respect to any property P.*

Corollary 2. *o_{add} is not necessarily an SMLO for any property, including Invariance and Response properties.*

Theorem 2. *For FSAs S and S' let $o : S \rightarrow S'$ be defined as $o : M(S) \rightarrow M(S')$ where $\exists z \in \mathcal{K}$ (the Boolean algebra), $z \neq 0$, $(v_1, v_2) \in E(S)$, such that $o(M(v_1, v_2)) = M(v_1, v_2) \vee z$. Then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$.*

Corollary 3. *o_{spec} is an SMLO for any property.*

Corollary 4. *o_{gen} is not necessarily an SMLO for any property, including Invariance and Response properties.*

The above theorems and corollaries cover the one-step operators. We next consider theorems that are needed to address the two-step operators. Although we found results for the one-step operators that were general enough to address *any* property, we were unable to do likewise for the two-step operators. Our results for the two-step operators determine whether these operators are necessarily SMLOs for Invariance or Response properties in particular. These results are quite intuitive. The first theorem distinguishes those learning operators that will satisfy Invariance properties from those that will not:

Theorem 3. *A machine learning operator is guaranteed to be an SMLO with respect to any Invariance property P if and only if $\not\uparrow G$ and $\not\uparrow L$ are both true.*

Corollary 5. *Operators $o_{delete+gen}$ and $o_{spec+gen}$ are guaranteed to be SMLOs with respect to any Invariance property.*

Corollary 6. *Operators $o_{delete+add}$, $o_{spec+add}$, o_{change} are not necessarily SMLOs with respect to Invariance properties.*

The next theorem characterizes those learning operators that cannot be guaranteed to be SMLOs with respect to Response properties.

Theorem 4. *Any machine learning operator that can introduce a new string with an infinitely repeating substring into the FSA language cannot be guaranteed to be an SMLO for Response properties.*

Corollary 7. *None of the two-step learning operators is guaranteed to be an SMLO with respect to Response properties.*

For those operators that do not have positive a priori results, we can still save time over total reverification by using incremental algorithms, which are described in the next section.

5.2 Incremental Reverification Algorithms

We just learned that operators o_{spec} and o_{delete} are "safe" learning operators (SMLOs), whereas o_{gen} and o_{add} are not. It is also the case that o_{gen} and o_{add} can cause problems (e.g., for Response properties) when they are part of a two-step operator. Therefore, we have developed incremental reverification algorithms for these two operators.

Recall that there are two ways that operators can alter accessibility: globally (G) or locally (L). Furthermore, recall that o_{add} can increase accessibility either way ($\uparrow$ G $\uparrow$ L), whereas o_{gen} can only increase accessibility locally ($\not\uparrow$ G $\uparrow$ L). We say that o_{gen} has only a "localized" effect on accessibility, whereas the effects of o_{add} may ripple through many parts of the FSA. The implication is that we can have very efficient incremental methods for reverification tailored for o_{gen}, whereas we cannot do likewise for o_{add}. This is also true for both two-step operators that have o_{gen} as their second step, i.e., $o_{delete+gen}$ and $o_{spec+gen}$. Because no advantage is gained by considering o_{add} per se, we develop incremental reverification algorithms for the most general operator o_{change}. These algorithms apply to all of our operators.

Here we present three incremental reverification algorithms – the first two are for execution after any instance of o_{change}, and the third is only for execution after instances of o_{gen} (or $o_{delete+gen}$ or $o_{spec+gen}$). These algorithms make the assumption that $S \models P$ prior to learning, i.e., any errors found from previous verification have already been fixed. Then learning occurs, i.e., $o(S) = S'$, followed by product re-formation, then incremental reverification (see Figure 1).

The first algorithm is an incremental version of $Total_{prod}$, called Inc_{prod}, which is tailored for re-forming the product FSA (step (6) of Figure 1) after o_{change} has been applied. Recall that in $SIT_{multplans}$ learning is applied to an individual agent FSA, then the product is re-formed. In all situations, the product must be re-formed with the negated property FSA after learning if the type of reverification to be used is AT. Algorithm Inc_{prod} assumes the product was formed originally using $Total_{prod}$. Inc_{prod} capitalizes on the knowledge of which single (or multi)agent state, v_1, and action, a, have their next state altered by operator o_{change}. (For simplicity, assume a is a multiagent action.) Since the previously generated product is stored, the only product FSA states whose next state needs to be modified are those states that include v_1 and transition on a.

After o_{change} has been applied, followed by Inc_{prod}, incremental model checking is performed. Our incremental model checking algorithm, Inc_{AT}, changes the set of initial states (for the purpose of model checking only) in the product FSA to be the set of all product states formed from state v_1 (whose next state was affected by o_{change}). Reverification begins at these new initial states, rather than the actual initial FSA states. This algorithm also includes another form of streamlining. The only transition taken by the model checker from the new initial states is on action a. This is the transition that was modified by o_{change}. Thereafter, Inc_{AT} proceeds exactly like $Total_{AT}$. Assuming $S \models P$ prior to learning, Inc_{AT} and $Total_{AT}$ will agree on whether $S' \models P$ after learning, whenever P is an Invariance or Response property [9].

We next present an incremental reverification algorithm that is extremely time-efficient. It gains efficiency by being tailored for specific situations (i.e., only in SIT_{1agent} or SIT_{1plan} when there is one FSA to reverify), a specific learning operator (o_{gen}), and a specific class of properties (Response). A similar algorithm for Invariance properties may be found in [10].

The algorithm, called Inc_{gen-R}, is in Figure 3. This algorithm is applicable for operator o_{gen}. However note that it is also applicable for $o_{delete+gen}$ and $o_{spec+gen}$, because according to the a priori results of Subsection 5.1 the first step in these operators is either o_{delete} or o_{spec} which are known to be SMLOs.

Assume the Response property is $P = \Box(p \rightarrow \Diamond q)$ where p is the trigger and q is the response. Suppose property P holds for plan S prior to learning, i.e., $S \models P$. Now we generalize $M(v_1, v_3) = y$ to form S' via o_{gen} $(M(v_1, v_3)) = y \vee z$, where $y \wedge z = 0$ and $y, z \neq 0$. We need to verify that $S' \models P$.

```
procedure check-response-property
if $y \models q$ then
    if ($z \models q$ and $z \models \neg p$) then output "$S' \models P$"
    else output "Avoid this instance of $o_{gen}$" fi
else
    if ($z \models \neg p$) then output "$S' \models P$"
    else output "Avoid this instance of $o_{gen}$" fi
fi
end
```

Fig. 3. Inc_{gen-R} reverification algorithm.

The algorithm first checks whether a response could be required of the transition condition $M(v_1, v_3)$. We define a response to be required if for at least one string in $\mathcal{L}(S)$ whose run includes (v_1, v_3), the prefix of this string before visiting vertex v_1 includes the trigger p not followed by response q, and the string suffix after v_3 does not include the response q either. Such a string satisfies the property if and only if $y \models q$ (i.e., for every atom $a \preceq y$, $a \preceq q$). Thus if $y \models q$ and the property is true prior to learning (i.e., for S), then it is possible that a response is required and thus it must be the case that for the newly added z, $z \models q$ to ensure $S' \models P$. For example, suppose a, b, c, and d are atoms, the transition condition y between STATE4 and STATE5 equals d, and $S \models P$. Let $\mathbf{x} = $ (a, b, b, d, ...) be an accepting string of S ($\in \mathcal{L}(S)$) that includes STATE4 and STATE5 as the fourth and fifth vertices in its accepting run. The property is $P = \Box$ (a $\rightarrow \Diamond$ d), and therefore $y \models q$ (because $y = q = $ d). Suppose o_{gen} generalizes M(STATE4, STATE5) from d to (d $\vee$ c), where z is c, which adds the string $\mathbf{x}' = $ (a, b, b, c, ...) to $\mathcal{L}(S')$. Then $z \not\models q$. If the string suffix after (a, b, b, c) does not include d, then there is now a string which includes the trigger but does not include the response, i.e., $S' \not\models P$. Finally, if $y \models q$ and $z \models q$, an extra check is made to be sure $z \models \neg p$ because an atom could be both a response and a trigger. New triggers are thus avoided. The second part of the algorithm states that if $y \not\models q$ and no new triggers are introduced by generalization, then the operator is "safe" to do. It is guaranteed to be safe ($S' \models P$) in this case because a response is not required.

Inc_{gen-R} is a powerful algorithm in terms of its execution speed, but it is based upon the assumption that the learning operator's effect on accessibility is localized, i.e., that it is o_{gen} with SIT_{1agent} or SIT_{1plan} but not $SIT_{multplans}$. (Recall that single agent o_{gen} may translate to multiagent o_{add} in the product FSA.) An important advantage of this algorithm is that it never requires forming a product FSA, even with the property. A disadvantage is that it may find false errors. In particular, if $S \models P$ prior to learning and if Inc_{gen-R} concludes that $S' \models P$ after learning, then this conclusion will be correct. However if Inc_{gen-R} finds an error, it may nevertheless be the case that $S' \models P$ [9]. Another disadvantage of Inc_{gen-R} is that it does not allow generalizations that add triggers. If it is desirable to add new triggers during generalization, then one needs to modify Inc_{gen-R} to call Inc_{AT} when reverification with Inc_{gen-R} fails – instead of outputting "Avoid this instance of o_{gen}." This modification also fixes the false error problem, *and* preserves the enormous time savings (see next section) when reverification succeeds.

5.3 Empirical Timing Results

Theoretical worst-case time complexity comparisons, as well as the complete set of experiments, are in [9]. Here we present a subset of the results, using Response properties. Before describing the experimental results, let us consider the experimental design. [3] The underlying assumption of the design was that these algorithms would be used in the context of evolutionary learning, and therefore the experimental conditions closely mimic those that would be used in this context. FSAs were randomly initialized, subject to a restriction – because the incremental algorithms assume $S \models P$ prior to learning, we restrict the FSAs to comply with this. Another experimental design decision was to show scaleup in the size of the FSAs. Throughout the experiments there were assumed to be three agents, each with the same 12 multiagent actions. Each individual agent

[3] All code was written in C and run on a Sun Ultra 10 workstation.

FSA had 25 or 45 states. A suite of five Response properties was used (see [9]). The learning operator was o_{change} or o_{gen}. Every algorithm was tested with 30 runs – six independent runs for each of five Response properties. For every one of these runs, a different random seed was used for generating the three FSAs and for generating a new instance of the learning operator. However, it is important to point out that on each run all algorithms being compared with each other used the *same* FSAs, which were modified by the *same* learning operator instance.

Let us consider Tables 1 and 2 of results. Table entries give average cpu times in seconds. Table 1 compares the performance of total reverification with the incremental algorithms that were designed for o_{change}. The situation assumed for these experiments was $SIT_{multplans}$. Three FSAs were initialized, then the product was formed. Operator o_{change}, which consisted of a random choice of next state, was then applied to one of the FSAs. Finally, the product FSA was re-formed and reverification done.

The method for generating Table 2 was similar to that for Table 1, except that o_{gen} was the learning operator and the situation was assumed to be SIT_{1plan}. Operator o_{gen} consisted of a random generalization to the product FSA.

The algorithms (rows) are in triples "p," "v" and "b" or else as a single item "v=b." A "p" next to an algorithm name implies it is a product algorithm, a "v" that it is a verification algorithm, and a "b" that it is the sum of the "p" and "v" entries, i.e., the time for *both* re-forming the product and reverifying. If no product needs to be formed, then the "b" version of the algorithm is identical to the "v" version, in which case there is only one row labeled "v=b."

Table 1. Average cpu time (in seconds) over 30 runs (5 properties, 6 runs each) with operator o_{change}.

	25-state FSAs	**45-state FSAs**
Inc_{prod} **p**	.000574	.001786
$Total_{prod}$ **p**	.097262	.587496
Inc_{AT} **v**	.009011	.090824
$Total_{AT}$ **v**	.024062	.183409
Inc_{AT} **b**	.009585	.092824
$Total_{AT}$ **b**	.121324	.770905

We tested the hypothesis that the incremental algorithms are faster than the total algorithms – for both product and reverification. This hypothesis is confirmed in all cases. All differences are statistically significant ($p < .01$, using a Wilcoxon rank-sum test) except those between Inc_{AT} and $Total_{AT}$ in Table 2. In fact, according to a theoretical worst-case complexity analysis [9], in the worst case Inc_{AT} will take as much time as $Total_{AT}$. Nevertheless, in practice it usually provides a reasonable time savings.

What about Inc_{gen-R}, which is even more specifically tailored? First, recall that Inc_{gen-R} can produce false errors. For the results in Table 2, 33% of Inc_{gen-R}'s predictions were wrong (i.e., false errors) for the size 25 FSAs and

Table 2. Average cpu time (in seconds) over 30 runs (5 properties, 6 runs each) with operator o_{gen}.

	25-state FSAs	**45-state FSAs**
Inc_{prod} **p**	.000006	.000006
$Total_{prod}$ **p**	.114825	.704934
Inc_{AT} **v**	94.660700	2423.550000
$Total_{AT}$ **v**	96.495400	2870.080000
Inc_{AT} **b**	94.660706	2423.550006
$Total_{AT}$ **b**	96.610225	2870.784934
Inc_{gen-R} **v=b**	.000007	.000006

50% were wrong for the size 45 FSAs. On the other hand, consider the maximum observable speedup in Tables 1 and 2. By far the best results are with Inc_{gen-R}, which shows a $\frac{1}{2}$-*billion-fold speedup* over $Total_{AT}$ on size 45 FSA problems! This alleviates the concern about Inc_{gen-R}'s false error rate – after all, one can afford a 50% false error rate given the speed of trying another learning operator instance and reverifying.

6 Applications

To test our overall framework, we have implemented a simple example of cooperating planetary rovers that have to coordinate their plans. They are modeled as co-evolving agents assuming $SIT_{multplans}$. By using the a priori results and incremental algorithms, we have seen considerable speedups.

We are also developing another implementation that uses reverification during evolution [17]. Two agents compete in a board game, and one of the agents evolves its strategy to improve it. The key lesson that has been learned from this implementation is that although the types of FSAs and learning operators are slightly different from those studied previously, and the property is quite different (it's a check for a certain type of cyclic behavior on the board), initial experiences show that the methodology and basic results here could potentially be easily extended to a variety of multiagent applications.

7 Related and Future Work

This paper has presented efficient methods for behavioral assurance following learning. The incremental reverification algorithms presented here are similar to the idea of local model checking [2] because they localize verification. The difference is that our methods are tailored specifically to learning operators. There is little in the literature about efficient model checking for systems that change. Sokolsky and Smolka [16] is a notable exception – especially since it presents a method for incremental reverification. However, their research is about reverification of software after user edits rather than adaptive multiagent systems.

There is a growing precedent for addressing multiagent coordination by expressing plans as automata and verifying them with model checking (e.g., [13], [4], [11]). Our work builds on this precedent, and also extends it – because none of this previous research addresses efficient *re*-verification for agents that learn.

Finally, there are alternative methods for constraining the behavior of agents, which are complementary to reverification and self-repair. For example, Shoham and Tennenholtz [15] design agents that obey social laws, e.g., safety conventions, by restricting the agents' actions; Spears and Gordon [18] design agents that obey physics laws. Nevertheless, the plan designer may not be able to anticipate and engineer all laws into the agents beforehand, especially if the agents have to adapt. Therefore, initial engineering of laws should be coupled with efficient reverification after learning.

Future work will focus on extending the a priori results to other learning operators/methods and property classes and other plan representations (such as stochastic/timed FSAs/properties), developing more incremental reverification algorithms, and exploring plan repair to recover from reverification failures [12].

Acknowledgements. This research is supported by ONR N0001499WR20010. I am grateful to Bill Spears for numerous useful suggestions.

References

1. Bäck, T. & Schwefel H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1).
2. Bhat, G. & Cleaveland, R. (1986). Efficient local model checking for fragments of the modal mu-calculus. *Lecture Notes in Computer Science*, 1055.
3. Büchi, J. (1962). On a decision method in restricted second-order arithmetic. *Methodology and Philosophy of Science, Proc. Stanford Int'l Congress.*
4. Burkhard, H. (1993). Liveness and fairness properties in multi-agent systems. *IJCAI'93.*
5. Clarke, E. & Wing, J. (1997). Formal methods: State of the art and future directions. *Computing Surveys.*
6. Courcoubetis, C., Vardi, M., Wolper, M., & Yannakakis, M. (1992). Memory-efficient algorithms for the verification of temporal properties. Formal Methods in Systems Design, 1.
7. Fogel, D. (1996). On the relationship between duration of an encounter and the evolution of cooperation in the iterated Prisoner's Dilemma. *Evolutionary Computation*, 3(3).
8. Freitas, Robert, Jr. (1999). Nanomedicine V1: Basic Capabilities. *Landes Bioscience Publishers.*
9. Gordon, D. (2000). Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, 13.
10. Gordon, D. (1998). Well-behaved Borgs, Bolos, and Berserkers. *ICML'98.*
11. Kabanza, F. (1995). Synchronizing multiagent plans using temporal logic specifications. *ICMAS'95.*
12. Kiriakos, K. & Gordon, D. (2000). Adaptive supervisory control of multi-agent systems. *FAABS'00.*

13. Lee, J. & Durfee, E. (1997). On explicit plan languages for coordinating multiagent plan execution. *ATAL'97*.
14. Proceedings of the Workshop on Multiagent Learning (1997). AAAI-97 Workshop.
15. Shoham, Y. & Tennenholtz, M. (1992). Social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73.
16. Sokolsky, O. & Smolka, S. (1994). Incremental model checking in the modal mu-calculus. *CAV'94*.
17. Spears, W. & Gordon, D. (2000). Evolving finite-state machine strategies for protecting resources. *ISMIS'00*.
18. Spears, W. & Gordon, D. (1999). Using artificial physics to control agents. *ICIIS'99*.

Ontology Negotiation in a Community of Agents
Extended Abstract

Sidney C. Bailin
Knowledge Evolution, Inc.

1 Background

Agents are now seen as playing an important role in realizing two key goals of technology development and deployment within NASA: reducing the lifecycle cost of missions, and enabling new mission concepts. Agents will contribute to the goal of *cost reductions* by removing the need for certain tasks to be performed by humans. *New mission concepts* will be enabled through the autonomous capabilities of spacecraft and instruments, and by the ability of scientists to obtain the information they need, seamlessly, when they need it.

We have been investigating the conditions required by knowledge communities of automated agents. In particular, we are focusing on the problems raised by differences in language or differences in ontology. Our assumption is that ontology conflicts will occur, frequently, in a world populated by agents. Agents will require a means to get to know each other, assess each other, and use the resulting knowledge as the basis for engaging in cooperative (or competitive) activity. A process of discovering or uncovering shared meaning is a prerequisite to the formation of agent-based knowledge communities.

Under what conditions do *strange agents* encounter each other and engage in a dialogue that might require negotiation of ontologies? What is the process by which agents discover opportunities for expanding their ontologies? We address these questions through the metaphor of a cocktail party in which strangers meet and learn something about each other. Although the metaphor is somewhat facetious, it allows us to define successive levels of mutual understanding that the participants in such a process can achieve:

1. Talk in distinct languages
2. Guess and confirm or clarify
3. Take note of the other agent's self-description
4. Continue dialogue to deepen mutual understanding
5. Engage the other agent in a task

The last of these steps is really the beginning of the next major phase of negotiation, after the initial meeting between agents. As an example of this task, we have used an information retrieval scenario: a space scientist working with a particular mission data archive intuits that data from another mission might be useful. Since the two missions have different scientific goals, the archives may be organized according to different conceptual schemes. We have identified a series of steps by which ontological mismatches may be discovered and resolved. In summary:

1. Respond to query with minimal effort
2. Express dissatisfaction with initial response
3. Enlist additional help
4. Identify ontological mismatch
5. Negotiate ontologies to construct integrated story
6. Evolve ontologies

This scenario raises some obvious technical challenges:

- How does the scientist's agent recognize the inadequacy of the first response?
- How does the scientist's agent construct an explanation of this problem?
- How does the first server infer from the explanation that another server can help?
- How do the two servers discover their conceptual mismatch?
- What properties of the ontologies will permit the conceptual reduction process described in stage 6?
- How do the two servers splice their common low-level concepts together to form an integrated shared ontology?
- What criteria are used to validate the integrated ontology?

It is in addressing these questions that the issues of formal languages arise.

2 Identifying and Resolving Ontology Conflicts

Ontological conflicts fall into one of the following categories:

- Unknown concepts
- Homonyms
- Synonyms

We treat these categories broadly, so that, for example, a concept that is used in a similar but slightly different way by two agents is considered an example of a homonym. In all of these cases, resolving the mismatch involves a dialogue between agents about the operational meaning of certain terms. The operational meaning of a term is the way in which the term's meaning is expressed through an agent's behavior. We assume that the operational meaning is embodied in an agent's rules for interpreting and acting on facts or beliefs involving the term. The rules may be explicit or implicit. Such rules do not, usually, completely characterize the intended meaning of a term. They represent a decision by the ontology developer to operationalize a portion of the meaning that is both significant and tractable.

There is a range of possible paths for inquiry starting from a conflict of operational meaning. Suppose that two or more agents suspect they may not mean the same thing by a particular term. Each agent articulates to the other(s) the rules that it uses to operationalize the term's meaning. They discover that each agent uses different rules. Since there is a potential conflict in operational meanings, several questions immediately arise:

1. Are the rules consistent with each other?
2. If they are consistent, are they relevant to each other? Or do they refer to completely different aspects of the term's meaning?
3. If they are inconsistent with each other, why is this?
4. If they are irrelevant to each other, why is this?

The process of ontology negotiation consists of recognizing the conflict, answering these questions, and deciding what to do about it. The same questions arise when two agents use different terms for the same concepts. In this case, each agent must guess at its equivalent to another agent's term. Suppose agents A_1 and A_2 conjecture that they use terms T_1 and T_2, to mean the same thing. In order to validate this conjecture, the agents must compare and contrast the operational meaning that each attributes to the term it uses. The rules that embody each term's operational meaning must be exchanged, and then the rules should be subjected to questions 1 through 4, above.

In order to answer these questions, the agents must be able to recognize when rules are consistent with each other (or not), and when they are relevant to each other (or not). Depending on the language, consistency may or may not be decidable; in any case, we are looking for constrained propositions that can be evaluated in a predictable amount of time. Similarly, relevance can be defined in principle as the existence of a chain of references linking the rules governing two terms. However, the practical application of this idea will probably involve setting pragmatic bounds on the length (or other properties) of the chains. Determining the contexts in which theorem proving techniques ca be practically used to answer questions 1 - 4, above, is one of the goals of our continuing research.

Now suppose that two agents have determined the consistency and relevance of their respective interpretations of a term. Or they have reason to believe that two terms (one in each of their respective vocabularies) mean the same thing. What then? We can identify four (not necessarily exclusive) possible outcomes:

1. Expand vocabulary to include new concept
2. Expand the operational meaning of a term
3. Explain differences in meaning
4. Model the divergence between ontologies

For example, in the case of synonyms, the two agents may come to an agreement on a standard terminology. Each agent need only supplement its ontology with a thesaurus structure containing the synonyms. Alternatively, an agent might revise its ontology to use the newly standardized terms. In the case of homonyms, the two agents may come to an agreement to make a distinction between, or qualification of, concepts that were previously considered identical. This may involve arbitrary changes to the structure of the ontology, e.g., morphological changes to the *is-a* tree.

3 Conclusion

Evolving an ontology in response to interactions with other agents requires several non-trivial capabilities:

- The ability to explain the operational meaning of expressions
- The ability to assess the relevance of another agent's use of an expression
- The ability to find a common shared vocabulary within which to negotiate
- The ability to talk about both agents' rules
- The ability to assess the consistency of another agent's rules with one's own
- The ability to modify the semantic structure of an ontology (e.g., the *is-a* hierarchy) while maintaining its integrity

Can we expect automated agents to be able to do these things? We can pose a converse question: can we call an automated agent "intelligent" if it cannot do these things?

There is an intrinsic obstacle to achieving these capabilities through symbolic programming techniques. The ability for agents to talk to each other about their rules assumes a common framework for talking about rules. At the very least, this includes a common notion of the structure of rules, e.g., a pair consisting of a set of pre-conditions and a set of actions. To assume such a framework seems contradictory to the goals of this investigation.

To mitigate this contradiction, we suggest that the framework be minimal and evolvable. We also observe that a common framework for talking about rules is not the same as a common ontology. It is, rather, a common meta-ontology. The assumption of a common meta-ontology is not immune to the problems posed in Section 1. However, it is less susceptible than the assumption that agents will share problem-level ontologies.

In light of this discussion, the need for dynamic ontology negotiation leads to the consideration of sub-symbolic methods. Such methods offer the ability of agents to learn, without pre-defining any domain-level or meta-ontologies. Could they be used by agents to evolve their ontologies in negotiation with other agents? There are precedents for two aspects of the challenge: deriving ontologies sub-symbolically, and using sub-symbolic methods in multi-agent learning. Our question is whether these two aspects can be united. This is an area for experimentation.

Analysis of Agent-Based Systems Using Decision Procedures

Ramesh Bharadwaj

Naval Research Laboratory
Washington, DC 20375-5320
ramesh@itd.nrl.navy.mil

In recent years, model checking has emerged as a remarkably effective technique for the automated analysis of descriptions of hardware systems and communication protocols. To analyze software system descriptions, however, a direct application of model checking rarely succeeds [1,3], since these descriptions often have huge (often infinite) state spaces which are not amenable to the finite-state methods of model checking. More important, the computation of a fixpoint (the hallmark of the model checking approach) is not always needed in practice for the verification of an interesting class of properties, viz, properties that are invariantly true in all reachable states or transitions of the system. To establish a property as an invariant, an induction proof, suitably augmented with automatically generated lemmas, often suffices.

Salsa is an invariant checker for specifications in SAL (the **S**CR **A**bstract **L**anguage). To establish a formula as an invariant without any user guidance Salsa carries out an induction proof that utilizes tightly integrated decision procedures, currently a combination of BDD algorithms and a constraint solver for integer linear arithmetic, for discharging the verification conditions. The user interface of Salsa is designed to mimic the interfaces of model checkers; i.e., given a formula and a system description, Salsa either establishes the formula as an invariant of the system (but returns no proof) or provides a *counterexample*. In either case, the algorithm will terminate. Unlike model checkers, Salsa returns a state pair as a counterexample and not an execution sequence. Also, due to the incompleteness of induction, users must *validate* the counterexamples. The use of induction enables Salsa to combat the state explosion problem that plagues model checkers – it can handle specifications whose state spaces are too large for model checkers to analyze. Also, unlike general purpose theorem provers, Salsa concentrates on a single task and gains efficiency by employing a set of optimized heuristics.

The design of Salsa was motivated by the need within the SCR Toolset [4] for more automation during consistency checking and invariant checking [1,3]. Salsa achieves complete automation of proofs by its reliance on *decision procedures*, i.e., algorithms that establish the logical truth or falsity of formulae of *decidable* sub-theories, such as the fragment of arithmetic involving only integer linear constraints called Presburger arithmetic. Salsa's invariant checker consists of a tightly integrated set of decision procedures, each optimized to work within a particular domain. Currently, Salsa implements decision procedures for propositional logic, the theory of unordered enumerations, and integer linear arithmetic.

After some experimentation, we arrived at the following practical method for checking state and transition invariants using Salsa (see Figure 1): Initially apply Salsa. If Salsa returns *yes* then the property is an invariant of the system, and we are done. If Salsa returns *no*, then we examine the counterexample to determine whether the states corresponding to the counterexample are reachable in the system. If so, the property is false and we are done. However, if one concludes after this analysis that the counterexample states are unreachable, then one looks for *stronger invariants* to prove the property. Salsa currently includes a facility that allows users to include such auxiliary lemmas during invariant checking. There are promising algorithms for automatically deducing such invariants, although Salsa currently does not implement them.

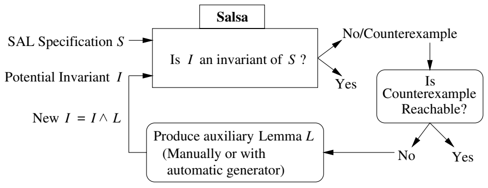

Salsa
SAL Specification S
Potential Invariant I
Is I an invariant of S ?
No/Counterexample
Yes
Is Counterexample Reachable?
New $I = I \wedge L$
Produce auxiliary Lemma L (Manually or with automatic generator)
No
Yes

Fig. 1. Process for applying Salsa.

In my future work, I would like to focus on using Salsa technology for descriptions of distributed and concurrent systems, most notably agent-based systems. My hope is that my interactions with scientists and researchers from the formal methods and agents based systems community will provide the necessary impetus for this work to proceed. The First Goddard Workshop on Formal Approaches to Agent-Based Systems is an excellent forum for such interactions.

References

1. Ramesh Bharadwaj and Constance Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.
2. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, LNCS 1785, Springer-Verlag, March 2000.
3. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.
4. Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

A Formal Approach to Belief-Goal-Role Agents

Walid Chainbi

ENIS
B. P. W - 3038 - SFAX - TUNISIA
Walid.Chainbi@fsegs.rnu.tn

Artificial intelligence has been largely characterized by attempts to understand and develop single agents which are abstract or concrete systems exhibiting some aspects of human intelligence. This approach has been insufficient due to the inevitable presence of agents in the real world. Accordingly, we must plan the activities of an agent while keeping in mind the other agents activities that either help or hinder him.

The recent years have witnessed a large interest in distributed artificial intelligence(DAI) with the goal of studying such type of *interaction*. The problem is no longer to analyze how an individual can achieve a task but how a group of individuals can solve complex problems and plan their actions collectively. In this respect, each agent of the organization has a limited knowledge and a simplified reasoning. Indeed, the interactions between agents become preponderant. Although there are situations where an agent can operate usefully by itself, the increasing interconnection and networking of computers is making such situations rare. Modern computing platforms and information environments are distributed, large, open, and heterogeneous.

Agents are being deployed in increasingly complex production environments such as avionics and factory control, where the failure or misbehavior of an agent might easily cause loss of life. Accordingly, a major challenge is to develop analysis tools and techniques for ensuring that agents will behave as we expect them to-or at least, will not behave in ways that are undesirable.

In this study, we present a theory for multi-belief-goal-role agents. Recall that agent theories are essentially specifications –agent theorists try to develop formalisms to formally represent and reason about the properties of agents. The proposed theory is a first-order, multi-modal and linear-time logic. As all the theories, it takes place within a conceptual context. In this respect, our approach is different from most known in DAI field: while the BDI architecture draws its inspiration from the philosophical theories of Bratman who argues that intentions as well as beliefs and desires play a significant and distinct role in practical reasoning, the presented approach stresses the interaction aspect to deduce the intentional structure of an agent. Indeed, starting from our study of cooperation in multi-agent systems, we identify the underlying concepts of an agent. These concepts consist of *beliefs* and *goals* as communication concepts and *roles* as concepts related to organization. Our approach draws its inspiration from the widely recognised fact that interaction is the most important single characteristic of complex systems. Many researchers believe that in future, computation itself will be understood chiefly as a process of interaction. This recognition may lead to the growth of interest in what we've adopted as a conceptual approach.

The underlying semantics of our theory are labeled transition systems which have been also used to describe the behavioral semantics of our agent model: each system is represented by a triplet including a set of states, a set of actions and a set of all possible system executions. An agent state is described by a triplet including beliefs, goals as communication concepts and roles as concepts related to organization. A transition consists of an execution step in the life-cycle of an agent.

Most known logical systems in DAI are based on a possible world semantics where an agent's beliefs, knowledge, goals, and so on, are characterized as a set of so-called possible worlds, with an accessibility relation holding between them. These models suffer from what Hintikka termed the logical omniscience problem: agents are so intelligent that they know all the logical consequences of their knowledge – an agent is not resource-bounded. Some alternative approaches have been adopted to avoid the problem of logical omniscience. A commonly known alternative is the *syntactic approach*, in which what an agent knows is explicitly represented by a set of formulae in its knowledge base. This set is not constrained to be closed under logical consequences or to contain all instances of a given axiom scheme. The *sentential approach* is more sophisticated than the syntactic approach, in that explicit beliefs are the primary beliefs and implicit beliefs are derived from them by closure under logical consequence. We adopt the syntactic approach to avoid omniscience.

Model Checking of Autonomy Models for an In-Situ Propellant Production System

Peter Engrand[1] and Charles Pecheur[2]

[1] NASA Kennedy Space Center, FL 32899 , U.S.A.
peter.engrand-1@ksc.nasa.gov
[2] RIACS / NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.
pecheur@ptolemy.arc.nasa.gov

Utilization of extraterrestrial resources, or In-Situ Resource Utilization (ISRU), is viewed as an enabling technology for the exploration and commercial development of our solar system. A key subset of ISRU is In-Situ Propellant Production (ISPP), which involves the partially autonomous production of propellants for planetary ascent or Earth return. To support the development and evaluation of ISPP technology, the NASA Kennedy Space Center is currently developing an ISPP hardware test bed for Mars missions, that uses carbon dioxide from the Martian atmosphere.

One of the challenges is the ability to maintain continuous plant operation without a Mars-based human presence, despite component failures and operational degradation. As a solution, KSC is employing the Livingstone controller to monitor an ISPP plant. Livingstone is a model-based autonomous health management agent developed at NASA Ames. It uses a model of the controlled system to track the system's behavior for anomalous conditions, diagnose component failures, and provide recovery recommendations.

Because this kind of intelligent systems address a very wide range of possible scenarios, A new approach to verification is needed to provide adequate confidence in their reliability. NASA Ames researchers, in collaboration with Carnegie Mellon University, have developed a translator that converts the Livingstone model and its expected properties into the input syntax of the SMV symbolic model checker. SMV then performs an exhaustive search of all possible executions of that specification and reports any property violation, illustrated by an execution trace that helps to localize the source of the violation.

These verification tools are used at Kennedy to validate the ISPP Livingstone models and provide feedback and suggestions to Ames on how to improve the tools. First experiments have shown that SMV can easily process the ISPP model and verify useful properties such as reachability of normal operating conditions or recoverability from failures. The translator and SMV have been used to verify expected global flow properties in a model representing a portion of the ISPP plant. The verification has pointed out a property violation, due to an improper modeling of flow equations. The latest version of the ISPP model, with 10^{50} states, necessitates some hand tuning of SMV optimizations but can still be processed in less than a minute, thanks to the power of BDD technology. This experience demonstrates how such verification tools can be used for interactive debugging as part of the design process.

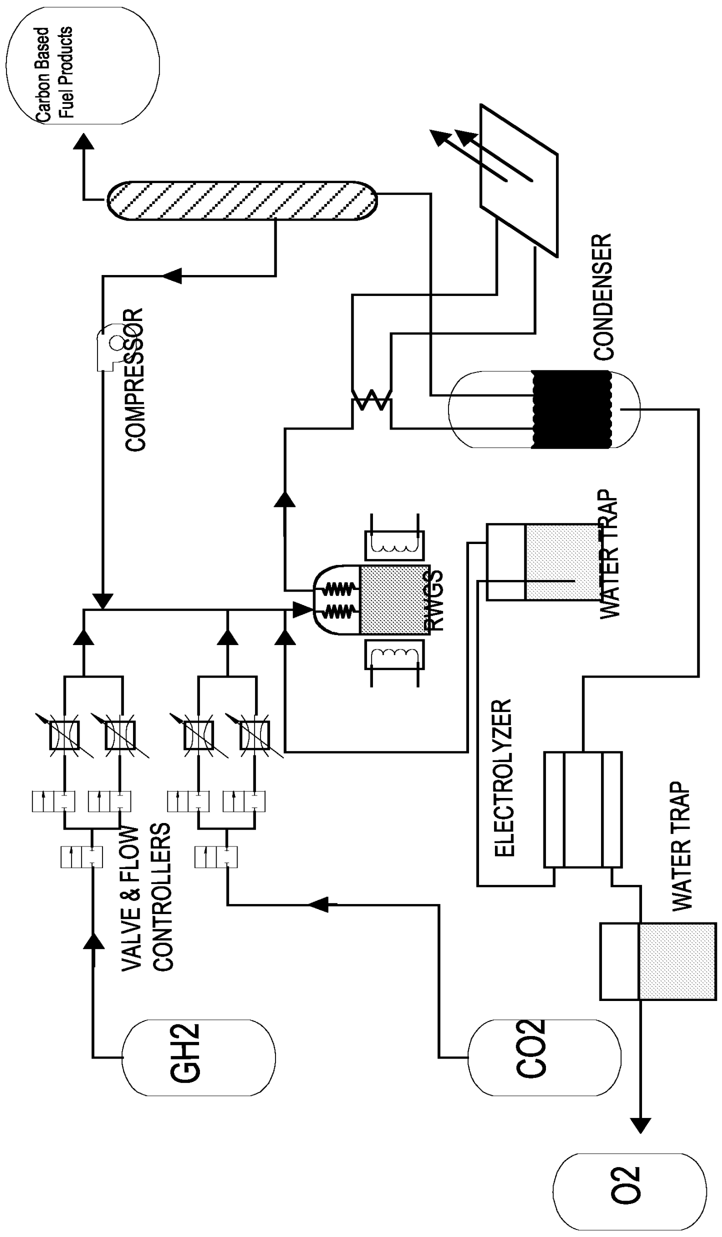


Fig. 1. Reverse Water Gas Shift schema

Adaptive Supervisory Control of Multi-agent Systems

Kiriakos Kiriakidis[1] and Diana F. Gordon[2]

[1] Department of Weapons and Systems Engineering, United States Naval Academy, Annapolis, MD 21402
kiriakid@novell.nadn.navy.mil,
http://wseweb.ew.usna.edu/wse/people/Kiriakidis
[2] AI Center, Naval Research Laboratory, Washington D.C. 20375
gordon@aic.nrl.navy.mil,
http://www.aic.nrl.navy.mil/~gordon

Multi-Agent Systems (MAS) provide for the modeling of practical systems in the fields of communications, flexible manufacturing, and air-traffic management [1]. Herein, we treat the general MAS as an Interconnected Discrete Event System (IDES). By design or due to failure, the actual system is often subject to change and so is its IDES model. For example, at the highest level, an existing subsystem may fail or a new subsystem may be connected. At a lower level, the structure of a subsystem may change as well. In spite of these changes, we want the MAS to preserve important properties which describe correct overall behavior of the system. For example, we want agent actions to be non-conflicting. Formal verification followed by self-repair is a solution, albeit a complex one. The complexity is due to the credit assignment, i.e., if verification fails we must determine which agents' actions are responsible for the failure of the MAS to satisfy the properties, and what is the best way to resolve the conflicts. This could entail highly complex reasoning to resolve.

A simpler solution may be found by recasting the multi-agent paradigm in the context of supervisory control theory – a method well-known in the realm of discrete event systems that is applicable to the class of IDES. Unfortunately, structural changes in an IDES hamper the application of current supervisory control theory. At present, the literature offers but a few works on adaptive or robust supervisory control to tackle the problem of uncertainty in the IDES model [2]. Here we present a novel approach that tackles the issue of adaptive supervisory control in a multi-agent context. Two architectures are presented.

The first of our two MAS architectures is one where a single supervisor guarantees the system-wide properties of the MAS. The second is a distributed architecture in which supervisors are themselves agents. For this second architecture, each supervisor agent supervises a MAS subsystem. In this latter case, there exist interactions between agents via subsystem interconnections.

Our objective is to develop a method for *adaptive* control of the aforementioned MAS architectures. The proposed approach combines supervisory control, learning, and verification as follows. Suppose that the supervisor controls the IDES so that it executes a desired language, which incorporates properties that the system needs to maintain regardless of structural changes. This desired language is in the form of a finite-state automaton (FSA). If a subsystem fails, the learning mechanism deletes from the desired language FSA events that pertain exclusively to the failed subsystem. Note that deleting events may fracture the desired language FSA into multiple isolated components. Therefore, to restore continuity in the FSA, an automaton repair algorithm is used. Then, model checking verifies that the resulting repaired FSA includes the aforementioned properties. If verification fails, the method applies a backup repair algorithm to the desired language FSA that requires no verification at the cost of yielding a more restricted desired language. In the end, the method automatically synthesizes a supervisor for the MAS from the new desired language, and closes loop around the IDES.

The advantage of this method is its simplicity. In particular, it obviates the need for a complex credit assignment process and instead substitutes a process of verification and repair to a single, typically small FSA – the desired language FSA. The complexity of translating repairs to the MAS is avoided. Instead we use the well-known process of automatic supervisor synthesis. Thus we obtain adaptability of the MAS via adaptive supervisory control.

Acknowledgements. This research is supported by the Office of Naval Research (N0001499WR20010 and N0001499WR20020) and the Naval Academy Research Council.

References

1. Jennings, N.,Sycara, K., and Wooldridge, M. (1998) A roadmap of agent research and development. Autonomous Agents and Multi-Agent Systems, **1**: 7–38.
2. Lin, F. (1993) Robust and adaptive supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, **38(12)**: 1842–1852.

Machine Learning for Logic-Based Multi-agent Systems

Eduardo Alonso and Daniel Kudenko

Department of Computer Science
University of York, York YO10 5DD
United Kingdom
{ea,kudenko}@minster.cs.york.ac.uk

1 Introduction

When developing a Multi-Agent System (MAS), it is very difficult and sometimes even impossible to foresee all potential situations the agents could encounter and specify their behaviour in advance. Therefore it is widely recognised that one of the more important features of high level agents is their capability to adapt and learn.

It is, however, worth noticing that whereas in some cases logic is used to specify the (multi-)agent architecture in order to incorporate domain knowledge, most learning algorithms that have been applied to agents are not logic-based, and instead other techniques such as reinforcement learning are used. Even though these techniques are successful in restricted domains, they strip agents of the ability to adapt in a domain-dependent fashion, based on background knowledge of the respective situation. This ability is crucial in complex domains where background knowledge has a large impact on the quality of the agents' decision making.

We propose to apply the logic-based learning techniques Explanation-Based Learning (EBL) [1] and Inductive Logic Programming (ILP) [2] to endow deliberative agents in multi-agent domains with learning capabilities. Specifically, our hypothesis is that *Logic-based learning techniques improve agents' performance in terms of effectiveness and adaptability in complex, dynamic MAS*. The justification for this hypothesis is as follows: (1) Agents in MAS domains need large amounts of knowledge (common-sense knowledge, domain specific knowledge, organizational knowledge); (2) the preferred representation for knowledge and reasoning is logic (because of its expressiveness, naturalness in representing declarative knowledge, and its well-known syntactical and semantical properties); (3) agents in dynamic domains have to learn new knowledge (and improve the existing one).

We have chosen *conflict simulations* as the domain to illustrate the potential of this approach. Conflict simulations are models of military confrontations and are an ideal testbed for adaptive logic-based MAS because of (1) availability of large amounts of crucial background knowledge, (2) diversity of underlying models which pose a challenge to generality and adaptivity of the MAS, (3) variations in complexity which allow to test the scalability of the system, (4) practical usefulness of intelligent computer opponents for military training and strategic decision making.

2 Logic-Based Learning Techniques

Roughly stated, learning in our MAS can be viewed as two different tasks: The first learning task (handled by EBL) puts the accent on improving the system's problem-solving capabilities and the second (handled by ILP) on the acquisition of new knowledge.

Explanation-Based Learning: Agents have to be able to refine their search control knowledge to construct and maintain an optimal organizational structure (*speed-up* learning). Obviously, distributed AI problem solvers, when presented with the same problem repeatedly, should not solve it the same way and in the same amount of time. On the contrary, it seems sensible to use general, structural knowledge to analyze, or explain, each training example in order to optimize future performance. EBL is a suitable strategy to implement this kind of learning. In short, EBL deduces general rules from single examples by generating an explanation of the examples and generalizing it. This provides a deductive method to turn first-principles knowledge into useful, efficient, special-purpose expertise.

Inductive Logic Programming: Inductive Logic Programming is a learning technique that given domain knowledge, a target predicate P, and positive and negative training examples of this predicate, computes a definition (or an approximation) of P in form of a logic program. For example, a useful target predicate could be `SuccessfulAttack(S,X,Y)` which predicts whether an attack of unit `X` on unit `Y` will be successful in state `S`.

In contrast to EBL methods, ILP computes a hypothesis not just based on simulation rules known beforehand but also on external and initially unknown circumstances such as the opponent's behaviour. Generally, relying on EBL-generated rules can turn out to be impractical in real-world domains in which agents work with incomplete knowledge, and thus ILP is an important addition to the system's effectiveness.

References

[1] T.M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning,* 1:4–80, 1986.

[2] S.Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming,* 19:629–679, 1994.

Using a Formal Framework for Agent System Design

V. Carchiolo, M. Malgeri, and G. Mangioni

Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
V.le A. Doria, 6 - I95125 Catania (ITALY)
{VCarchiolo, MMalgeri, GMangioni}@diit.unict.it

1 Introduction

The implementation of mobile agent systems involves great problems in particular it is often hard to guarantee that the specification of a system that has been designed actually fulfils the design requirements. Especially for critical applications, for example in real-time domains, there is a need to prove that the system being designed will have certain properties under certain conditions (assumptions).

These elements have induced researchers to explore new alternatives. One alternative to traditional techniques is to use formal methods in several stages of design. In mobile agent system design the term formal methods refers to the use of mathematical methods for the specification, validation and simulation of systems.

Using traditional methods, the design of a agent system requires three main phases: specification of the requirements, implementation of the device, and validation. The last phase consists of checking that the final device meets the specifications. This step normally involves testing the implementation by simulation. As exhaustive testing is not feasible, the implementation is only tested in a fraction of the states it could possibly find itself in, so correct behaviour can only be ensured for the test patterns actually tested. An alternative approach is to use formal methods in the validation stage. Formal Verification uses mathematical tools to check that the implementation meets the requirements expressed in the specifications, making it possible to run exhaustive tests on the final device. This way of proceeding requires formal descriptions for both specification and implementation.

There are various methods for the formal specification of a distributed systems; the most commonly used can be grouped into two classes:

– Methods using temporal logic to express the properties a system should have (CTL, LTL, μ Calculus, ...)[1].
– Methods in which specification is made in terms of a high-level model of the system (First-order predicate logic, CSP, UNITY, ...)[1].

The techniques currently used for formal verification can be grouped according to the varying degrees of automation they offer, as follows:

– Model Checking Tools
– Interactive Theorem-Proving Tools
– Tools integrating Model Checking and Theorem-Proving.

With Model Checking Tools verification is totally automatic, but it is affected by an explosion of states when dealing with systems of a certain size [2].

Exploiting the mathematical bases of the formalisms used to describe the specifications and the final device, checking that an implementation meets specifications boils down to a mathematical demonstration. Theorem-proving tools are systems that assist such demonstrations, interacting with the designer and automating the application of theorems. Their advantage lies in the possibility of verifying even systems of large dimensions. The disadvantage is that verification with these tools is not totally automatic [2].

The third category of tools combines the features of Model Checking with those of Theorem- Proving, thus making it possible to check medium-size designs automatically. We spend some time discussing why some of the basic assumptions of the p-calculus and of other concurrent formalisms do not satisfy our particular needs.

Some early paradigms for concurrency, such as Actors [3], allowed highly dynamic systems. However, the main formalized descriptions of concurrency began by considering only static connectivity. This is the case for Petri Nets [4], for CSP [5], and for CCS [6]. In CCS, in particular, the set of communication channels that a processes has available does not change during execution. In some versions of CSP (and in Occam [7] the set of processes cannot change either. The p-calculus has no inherent notion of distribution. Processes exist in a single contiguous location, by default, because there is no built-in notion of distinct locations and their effects on processes. Interaction between processes is achieved by global, shared names (the channel names); therefore every process is assumed to be within earshot of every other process. Incidentally, this makes distributed implementation of the original p-calculus (and of CCS) quite hard, requiring some form of distributed consensus.

In this paper we propose a design flow based on the formal description technique `LOTOS` [8][9], which assists the designer of mobile agent systems during the various phases of the design flow. The approach is based on using a `LOTOS` specification style more suitable for modeling process migration (similar approach has been followed in [16]). `LOTOS` is a formal description technique standardized for open system specification. It is based on Milner's Calculus of Communicating Systems (CCS) [6] and Hoare's Communicating Sequential Processes (CSP) [5] models using ACT ONE [10] to data types definition.

2 Agents

Mobile agents are active objects that have a behaviour, a state and a location. An agent can be seen as a set of data, a code and an execution context that can migrate through a network during execution. Mobile agents are autonomous because once called they decide autonomously where to migrate to and what code to execute. This behaviour is implicitly defined by the agent code [11]. An agent reacts to stimuli from the outside environment and can be persistent in the sense that its execution can be interrupted and the data stored so as to be re-used when the agent wakes up again, even in a different location. The use of a mobile code is extremely important as it offers the following facilities:

– software distribution on demand;
– management of asynchronous tasks;
– reduction in communication costs.

There are several code mobility modes:

1. *Remote Execution and Code on Demand*: the agent is transferred before execution of the associated code so both code and data are transferred. This scheme is followed by Tacoma [12], whereas the Code on Demand scheme is used to manage Java Applets [13].
2. *Weak Migration*: only code + state of data are transferred. This is the scheme used by Aglets and Mole.
3. *Strong Migration*: code + data + state of execution are transferred. This scheme is the most complete one and is used for AgentTcl [11], Telescript [14] and Odissey [15].

In reality the Remote Execution and Code on Demand schemes do not allow mobile agents to be managed as well as the weak and strong migration schemes do. Choice between the latter, which have the same power of expression, depends on implementation features.

All the existing engines shares the same structure: the agents communicate are able to communicate mong them and to move from one place to another: so we define the following terms:

1. place is the location where the agent may be executed, usually it coincides with an host,
2. engine or region, is the logical environment where the agents live, therefore all agents can move, excute or communicate only inside their engine
3. agent, is the core of system, usually there exists some stanzial agents and some moving agents

3 LOTOS

LOTOS Language Of Temporal Ordering Specification)[8] which has been standardized by ISO (International Standards Organization) between 1981 and 1988

The basic idea is that the behavior of a system can be described by observing from the outside the temporal order in which events occur. In practice, the system is seen as a black-box which interacts with the environment by means of events, the occurrence of which is described by LOTOS behavior expressions. The language has two components: the first is the description of the behavior of processes and their interaction, and is mainly based on the CCS[6] and CSP[5]; the second is the description of the data structure and expressions, and is based on ACT ONE[10]. In the field we are applying the language the data component is almost trivial thus we will discuss only the behavioral part of LOTOS . In LOTOS distributed systems are described in terms of processes; the system as a whole is represented as a process, but it may consist of a hierarchy of processes (often called subprocesses) which interact with each other and the environment. LOTOS models a process only describing it through its interaction with the environment. The atomic forms of interaction take the name of events. The definition of a process in LOTOS is:

```
process <proc-id> <par-list> := <behaviour-expression>
endproc
where:  <proc-id>    is the name to be assigned to the process;
        <par-list>   is the list of events with which the
                     process can interact with the environment;
```

<behaviour-expression> s are the LOTOS expressions which define the behaviour of the process The recursive occurrence of a process-identifier in a behaviour expression makes it possible to define infinite behavior (both auto- and mutual recursion are possible).

In the following we describe the basic operators by which it is possible to describe any system. A completely inactive process, i.e. one which cannot execute any event, is represented by stop.

The action represents the basic synchronization. This operator produces a new behavior expression from an existing one, prefixing it with the name of an event. If B is a behavior expression and a is the name of an event, the expression a;B indicates that the process containing it first takes part in the event a and then behaves as indicated by the expression B.

The choice operator models the non-deterministic behaviors which takes place when two (or more) event are available. If B1 and B2 are two behavior expressions, then B1 [] B2 denotes a process which can behave both as B1 and as B2 . Choice between the two forms of behavior is made by the environment.

To simplify the description of the system being specified a lot of derived operator are present in LOTOS . They can be easily rewritten in term of the basic one but the specification result longer and difficult to understand. Some of the derived operators are described in the following.

The arbitrary interleaving represents the independent composition of two processes, B1 and B2 and is indicated as B1 ||| B2 . If the two processes have some event in common, B1 ||| B2 indicates their capacity to synchronize with the environment but not with each other.

The parallel operator is indicated as B1 || B2 and it means that the two processes have to synchronize with each other in all events. B1 || B2 can take part in an event if and only if both B1 and B2 can participate.

The general parallel composition is a general, way of expressing the the parallel composition of several events and is denoted with the expression
`B1 |[a`$_1$`,...,a`$_n$`]| B2` .

The sequential composition of two processes, `B1` and `B2` , is indicated as `B1>>B2` and model the fact that when the execution of `B1` terminates successfully `B2` is executed (”`>>` ” is also known as an enabling operator). To mark successful termination, there is a special `LOTOS` process called `exit` .

Table 1. Type of interactions among processes

process B_1	process B_2	sync condition	int. sort	effect
$g!E_1$	$g!E_2$	value(E_1) = value(E_1)	matching	synchronization
$g!E$	$g?x : t$	value$(E) \in$ domain(t)	passing	after synch. x = value(E)
$g?x : t$	$g?y : u$	$t = u$	generation	after synchronization $x = y = v,\ \forall v \in$domain$(t)$

The introduction of types makes it possible to describe structured events. They consist of a label or gate name which identifies the point of interaction, or gate (i.e. an event), and a finite list of attributes. Two types of attributes are possible: value declaration and variable declaration. Table 1 presents the permitted interactions.

4 Using LOTOS to Specify Agents

We propose to use the background of communication protocols development and, in particular, the experience in protocol specification to specify Mobile agent systems, and we chose `LOTOS` as specification language.

In fact, the communication primitives among agents mirror the characteristics of the communication protocol primitives that aim at solving the synchronization and data exchange matters in a distributed environment. Mobile agents must manage both communication and mobility problems.

In this paper is presented an approach based on `LOTOS` to model both communication and mobility. `LOTOS` is able to model agent with a high level of abstraction where only the communication aspects are modelled, whereas `LOTOS` do not provide an adequate support for mobility. Our work aims at defining a design flow that starting from the agent behaviour specification permits to introduce step by step the complexity linked with the mobility modelling and with the agent locality. During the first steps it is important to describe what the agent must do and not how the agent work, that is the mobility aspect are hiding.

`LOTOS` do not have a direct support for mobility due to the static process definition, than the agent behaviour is emulated through some precise rules to define the agent.

The development of an agent-based application (sometime mobile agents) can be divided in three phases; each of them adds a detail at the specification and then reduces the abstraction degree:

1. *higher level of abstraction:* specify the agent needed to implement the communication protocol;
2. *medium level of abstraction:* the mobility details are introduced but a unique execution environment are considered;
3. *lower level of abstraction:* the agent location specification are introduced, that is, the fact that each agent are executed in a different location is modelled;

Previous steps allows us to pass from an abstract to a concrete specification; each of these steps is independent from the agent engine, therefore the specification obtained following this flow can be implemented using several agent engine like, for example, an algorithm is independent by the programming language used to implement the algorithm.

4.1 Agents Co-operation

During the first phase are identified the target application behaviour and the communication protocol among the agents. Generally, in this phase are identified some server agents and others agents communicating with server to elaborate the provided data. In this step of the design the application is viewed as a set of co-operating agent without the definition of the location or mobility aspects of each agent. The communicating process description is the main aspect covered by `LOTOS` semantics, and then the agent specification is easily and direct.

`LOTOS` semantics is based on description of communicating process, thus the cooperation between of agent can be specified in a natural way. Some difficults arises when a `LOTOS` parallel operator composes the agents in order to obtain the global behaviour of the application. In fact, the `LOTOS` synchronisation semantics require that all the processes having the same gate must be ready to rendezvous. This fact can produce a deadlock situation. Before discussing the problem of synchronisation among processes (and then among agents), we analyse the synchronisation protocol and the data exchange between two agents. To enable the information exchange between agents they must have a common gate (in the following we call this gate `comm` with the meaning of communication gate) through the data are exchanged.

For example, a typical action of a service request can be described by:

```
comm!data
```

If we consider an agent that receives a service request (when only two agents there exists) it can be described by:

```
comm?data:DATATYPE
```

Let us note the unique condition to synchronise the data exchange is that both agents must have the same gate `comm.`

Unfortunately, the definition of a synchronisation gate for each pair of agents is not feasible, in fact:

1. the agent specification cannot be independent, to add a new agent the agent specification must be modified;
2. the gate number exploits and the specification can be unreadable.

To overcoming these problems we will use only a synchronisation gate called `comm` for all agents and we will define an ad-hoc communication protocol having the following features:

1. each agent must send data, destination agent name and source agent name;
2. each agent would be always quite to synchronise itself through `comm,` the undesired data will be put in a garbage collector;
3. each agent must be provide of a synchrony component used to hear other agents and an asynchrony component to do what it must do.

According to this approach a general service request must be have the following form:

```
comm!destName!myName!data
```

where `destName` is the name of the destination agent name `myName` is the name of the current agent.

In the same way an agent that receives a service request can be described by:

```
comm!myName?sourceName:AgentName?data:DATATYPE
```

The data exchange is enabled when all the parameter names agree. Finally an application based in co-operating agents cam be specifies as follows:

$$\texttt{APPLICATION := AGENT1(Name}_1\texttt{,comm) || . . . || AGENTn(Name}_n\texttt{,comm)}$$

Where $\texttt{Name}_i$ is the name assigned to the agent.

Each agent is modelled as follows([16]):

$$\texttt{AGENT}_i\texttt{ := MYJOB(Name}_i\texttt{,comm) ||| LISTEN(Name}_i\texttt{i,comm)}$$

where the first, viz. `LISTEN,` hears at the gate `comm;` the second, named `MYJOB` specifies the behaviour of the agent. `LISTEN` guarantees the correct synchronization through the gate `comm` and throws in the garbage collector all the undesidered data.

4.2 How Adding Mobility to Agent Specification

LOTOS does not support, unlike other process algebra, the process mobility, but it is possible to model the process migration through different location using another gate called m . Through gate m the agent receives the name of the current location or sends the name of the destination location. When we provide the specification at a high level of abstraction (phase 1 of the design flow) the location are described as an unique process named REGION that models the agent engine. Process REGION manages the:

1. message routing between source and destination agents;
2. notify agent to the current location.

Introducing the concept of region allows us to model agent migrations. In fact, our believe it is that at a high level of abstraction the agent specification only requires the description of agent cooperation (synchronization and data exchange). In the other design phases (medium and low level of abstraction) it is needed to introduce the agent migration facility in order to take into account the migration impact in the system performance and to evaluate the security aspects. In the specification at a medium level of abstraction (phase 2 of the design flow) we models agent migration without taking into account the location. Then the system where migration is taken into account is like the following:

AGENT1(Name$_1$,comm) || . . . || AGENTn(Name$_n$,comm) || REGION(m,comm)

The proposed specification style and the hiding LOTOS feature allows us to blind unimportant actions. It is possible to proof the previous process is equivalent to agent specification composition hiding gate m.

4.3 More on Agent Specification

To introduce the place concept we will follow the same approach. Place specification permit to analize the problems of management locations. In this case the system is specified as follows

AGENT1(Name$_1$) || PLACE(m$_1$,comm$_1$,m$_i$,comm$_i$ || $\cdots$

The main matter during this specification stage is to model the intra-place communication. Due to LOTOS characteristics the full synchronization among places must be taken into account.

This specification level, highlighting implementation details dealing with place (often mapped one-to.-one with hosts) permits designer to model problems connected with addressing, naming, transmission delay, etc.

In summary we have the folloeing three view of the application (going from more abstract specification to lower one):

1. only agent are specified :

LEVEL1 := AGENT1(Name$_1$) || . . . || AGENTn(Name$_n$)

2. modelling migration, thus REGION is taken into account

   $$\texttt{LEVEL2 := AGENT1(Name}_1\texttt{) || . . . || AGENTn(Name}_n\texttt{) || REGION(m,comm)}$$

3. at least also place are modelled

   $$\texttt{LEVEL3 := AGENT1(Name}_1\texttt{) || PLACE(m}_1\texttt{,comm}_1\texttt{,m}_i\texttt{,comm}_i\texttt{ ||} \cdots$$

4. considering the appropriate equivalence relation the following holds:

   $$\texttt{LEVEL1} \approx \texttt{(hide m in LEVEL2)} \approx \texttt{(hide m, m}_i\texttt{ in LEVEL3)}$$

5 Conclusions

We apply our specification style in the design of Virtual Serice Provider (VSP) [17]. VSP consists of a pool of heterogeneous hosts that clients see as a single server. VSP is based on a mobile agent technology using multicast addressing. A client requiring a service transmits his request to the group address, not knowing how many servers make up the VSP, where they are locale or which of them will actually provide the service. In fig 1 is shown the client point of view of vsp.

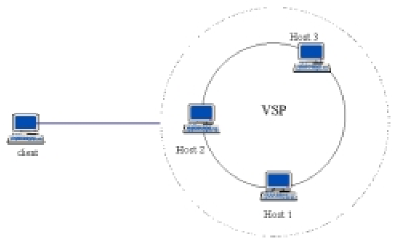


Fig. 1. VSP

Present work aims to use a well-known technique suitable to describe communication between processes to mobile agents technology. We achieve the goal by using an appropriate specification styles, that permits deisgner to isolate the main aspects of the agents: communication, migration and location. Tha approach has been tested designing a large application which required many cooperating agents and the definition of complex synchronization protocols.

References

1. E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research,* 28(4):626–643, December 1996.

2. Gerth R. Kelb P. Dams, D. Practical symbolic model checking of the full calculus using compositional abstractions. Technical Report Report, Eindhoven University of Technology, Department of Mathematics and Computer Science, 1996.
3. G. A. Agha. Actors: a model of concurrent computing in distributed systems. *iMIT Press.*
4. W. Brauer. Net theory an applications. In *Proceedings of the Advanced Course on General Neth Theory of Processes and Systems,* Hamburg, 1980. Springer LNCS.
5. C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.
6. R. Milner. *A Calculus of communicating systems.* LCNS 92. Springer Verlag, New York, 1980.
7. INMOS Ltd. Occam programming manual. *Prentice Hall,* 1984.
8. ISO-IS-8807. *Information Processing Systems, Open System Interconnection, LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO,* June 1988.
9. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISD Systems,* 14:25–59, 1987.
10. B. Mahr, H. Ehrig. *Fundamentals of Algebraic Specifications.* 1 EATCS Monographs on Computer Science. Springer-Verlag, 1985.
11. R. S. Gray. AgentTCL: A Transportable Agent System. In *Proceedings of CIKM – Workshop on Intelligent Information Agent,* 1995.
12. D. Johansen, R. van Renesse, and F. B. Schneder. An Introduction to TACOMA Distributed System Version 1.0. Technical Report 95–23, University of Tromso, 1995.
13. The java WEB pages. URL: http://www.javasoft.com.
14. General Magic Inc. The telescript language reference. URL: http://www.genmagic.com/telescript/TDE/TDEDOCS HTML/telescript.html.
15. General Magic Inc. Odyssey web site – URL: http://www.genmagic.com/agents.
16. Najm E. and J.B. Stefani. *Dynamic configuration in LOTOS.* K. R. Parker and G. A. Rose editors – Formal Description Techniques IV. North-Holland, 1991.
17. V. Carchiolo, M. Malgeri, and G. Mangioni. An agent based platform for a service provider. In *Proceedings of 2nd IMACS International Conference on Circuits, Systems and Computers (CSC98),* iTerma Hatzikyriakou Piraeus (Greece), 1998.

Modeling Agent-Based Systems

Sanda Mandutianu

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109-8099
sanda.mandutianu@jpl.nasa.gov

The agent-based systems have emerged from the necessity to manage complexity. Building reliable, maintainable, extensible, and re-usable systems that conform to their specifications requires modeling techniques that support abstraction, structuring, and modularity. The most widespread methodologies developed for the conventional software systems are various object-oriented approaches. They have achieved a considerable degree of maturity and are supported by a large community of software developers. The system architecture of object-oriented systems is based on the notion of objects, which encapsulate state information as data values and have associated behaviors defined by interfaces describing how to use the state information. Object oriented formal approach address almost all the steps in the process of designing and implementing a software system, providing a uniform paradigm across different system scales and implementation languages.

The agent paradigm, at the cross-road between conventional software, AI (Artificial Intelligence) and DAI (Distributed Artificial Intelligence) has shifted from the object-centric approach towards more ambiguously described entities, capable of autonomous behavior, embedded in changing, uncertain worlds, reacting to the changes in flexible ways. Object oriented methodologies are not completely applicable to agent-based systems, although these systems can be partially described by them (which in fact contributed to the confusion). As it is right now, it seems that the existing formalisms for describing and reasoning about agents have failed to provide full support for the process of agent design.

Any formal approach has eventually to materialize into a design methodology. A methodology to support the design and specification of agent systems should provide a conceptual framework for modeling systems that are decomposed into agents as entities characterized by their purpose, their capabilities or expertise, and their roles in the interactions that may occur between them. In addition, certain assumptions have to be made about the internal structure of the agents. In some cases, the agents conform to the same modeling approach (i.e. with the BDI – Belief, Desire, Intention model, the basic notions are beliefs, goals, plans), or in a weaker approach, the agents must conform only to an interaction protocol. Unlike object-oriented methodologies, the primary emphasis is on abstractions such as roles, interaction, or goals.

One can observe that even the simpler notions of design, plan, pattern, can be defined over and over again, at different levels. One can plan for the design, and the execution of these plans can result in a design, etc. Nevertheless, there is something at a more fundamental level about design in general. Formal models for design can be defined using general notions. Ontologies and engineering models impart local order in an accessible form and help to put the models into a standard form that can be incorporated into design environments. One can develop useful ontologies and protocols for representing and acquiring process information. Using languages such as UML one can at least partially design agent-based systems.

There appears to be increasing interest in flattening control systems such that a community of cooperating agents can coordinate their activities to achieve overall decision making processes via negotiation. It seems appropriate for multi-layered approach to decision-making processes. An agent is considered as an entity described in terms of common sense modalities such as beliefs, commitments, intentions, etc. This intuitive approach has its benefits for complex systems such as distributed systems, robots, spacecraft, etc., where simpler, mechanistic descriptions might not exist. The agent might also have a repository of recipes (plans or rules) which specify the course of actions that has to be followed by the agent to achieve its intentions. The beliefs are updated from observations of the environment and the effect of the interactions with other agents.

The agent behavior is expressed in a declarative agent definition language, which is used by an interpreter, which acts as a controller of the agent actions. The agent execution cycle consists of the following steps: processing new messages, determining which rules are applicable to the current situation, executing the actions specified by these rules, and possibly, planning new actions.

In order to achieve coordination in multi-agent systems the agents might have to negotiate, they have to exchange information, i.e. they need to communicate. The agents have the ability to participate in more than one interaction with another agent at the same time. The structure to manage separate conversations is the protocol. The protocol provides additional structured contextual information to disambiguate the meaning of the message. In these conversations, an agent can play different roles depending on the context of the tasks or subjects. A role is the defined pattern of communications of an agent when implementing a protocol. An agent can assume several roles, when appropriate, depending on the protocol.

The interactions happen at the level of goals and commitments, rather than explicit actions. Thus, an agent can communicate its objectives, and the recipient agent can do the reasoning and planning accordingly. A simple descriptive language for goals has been defined and has been used as the chosen content language. The agents are heterogeneous and distributed. This means that they reside on different platforms and have different domains of expertise. In the current implementation, the interoperability is achieved by the system infrastructure. Agent interoperability is realized at two levels: knowledge level and interoperability mechanisms represented by the current distributed objects middleware (i.e. CORBA, RMI). At the knowledge level the agents share their knowledge. They do this by interacting at run-time, based on a common set of models, grouped in a shared ontology. The substrate of this process is the concept of virtual knowledge; each agent must comply with the knowledge base abstraction when it communicates with other agents. This technology allows the agents to interact at the knowledge level, independently of the format in which the information is encoded. The ontology represents the formal description of the model of the application domain. The model specifies the object in the domain, operations on them and their relationships.

Given the complexity of formally specifying agent based systems as described above, the use of weaker formal approaches are still acceptable and actually represents the current norm. Nevertheless, some topics taken in isolation can be addressed using formal methods, such as the BDI formalism (although the existing implementations usually are not exactly representing the formalism). We used formal methods to define ontologies for instance, and some other less formal approaches such as design patterns to specify system behaviors, using UML to define different design models.

References

1. Jennings, N. R., Sycara, K., Woolridge, M.: A Roadmap of Agent Research and Development. In Autonomous Agents and Multi-Agent Systems, 1, 275-306 , Kluwer Academic Publishers, Boston, 1998.
2. Labrou, Y., Finin, T., Peng, Y.: The Interoperability Problem: Mobile Agents and Agent Communication Languages. In Proceedings of HICCS 32, the 32th Hawaii International Conference on System Sciences, Jan. 1999.
3. Mandutianu, S., Cooperative Intelligent Agents for Mission Support. In Proceedings of IEEE Aerospace Conference, March 1999.
4. Mandutianu, S., Stoica, A.,: An Evolvable Multi-Agent Approach to Space Operation Engineering. In Proceedings of the International Conference on Multi-Agent Systems, July 1999.
5. Rao, A., Georgeff, M.: BDI Agents: From Theory to Practice. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, June 1995.

Modeling Agent Systems by Bayesian Belief Networks

Yun Peng

Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
ypeng@cs.umbc.edu

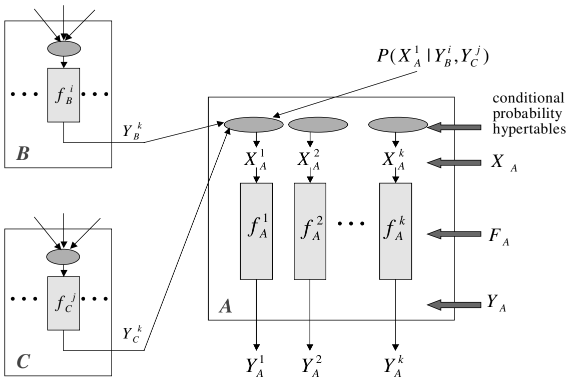
Development of multi-agent system (MAS) applications is often complicated by the fact that agents operate in a dynamic, uncertain world. Uncertainty may stem from noisy external data, inexact reasoning such as abduction, and actions by individual agents. Uncertainty can be compounded and amplified when propagated through the agent system. Moreover, some agents may become disconnected from the rest of the system by temporary or permanent disability of these agents or their communication channel, resulting in incomplete/inconsistent system states. How should we represent individual agents acting in such an uncertain environment, and more importantly, how can we predict how the MAS as a whole will evolve as the result of uncertain inter-agent interactions? These questions cannot be correctly answered without a correct agent interaction model based on a solid mathematical foundation.

In the past decade, Bayesian belief networks (BBN) have been established as an effective and principled general theoretical framework for knowledge representation and inference under uncertainty. It can be shown that a BBN can support any probabilistic inference about the system it models through Bayesian conditioning. However, classic BBN is inadequate in modeling MAS for the following reasons.

- BBN represents the world at very fine a level of granularity, where nodes, which correspond to random variables of interest, have no internal structure; and the variables may associate with each other only by direct causal links, whereas agents are *aggregate* units of multiple variables which associate with each other and with variables in other agents by potentially very complex functions and interactions.
- BBN does not provide means to explicitly represent actions. It is theoretically incorrect to represent actions as observations of some exogenous variables, as suggested by some people, because the influence of any observation can be computed from the joint distribution while external actions tend to *change* the existing distribution. Actions are particularly important in MAS since these include **communicative acts** that form the basis of much of MAS dynamics.
- Existing inference methods with BBN assume consistency of the states of the system. They do not support scenarios where variables are instantiated incompletely or inconsistently in the system. .

We plan to address these issues rigorously by extending the classic BBN formulation to MAS. The result we envision would be a **hypernetwork** in which agents are represented as **hypernodes** with potentially complex internal structures and functionality, including the actions the agent can take. The hypernodes are connected by **hyperlinks,** which represent the complex interaction patterns between agents so that the interdependence between variables involved in inter-agent interactions can be easily and clearly specified. The degree of uncertainty associated with hyperlinks will be represented by **conditional probability hypertables** at each hypernode, which can be viewed as a high-dimensional composition of the conditional probability table in classic BBN. The hypernetwork structure and the hyper-tables together compactly encode the joint probability distribution of the system's variables, and are thus capable of supporting provably correct inferences of MAS in the probability theoretic sense.

The figure below illustrates what hypernodes may look like and how they may be connected by hyperlinks. An agent, represented by a hypernode, can be defined by a 3-tuple (In_A, Out_A, F_A), where In_A is the set of all variables inputted to Agent *A* from other agents, Out_A is the set of variables emitted from *A,* and F_A is the set of functions that map variables in In_A to variables in Out_A. The details of these functions and their internal variables are shielded, except that we assume any $f_A^i \in F_A$ is a stochastic function that gives the conditional probability distribution $f_A^i(X_A^i, Y_A^i) = P(Y_A^i \mid X_A^i)$, where $Y_A^i \subseteq Out_A$ and $X_A^i \subseteq In_A$. A hyperlink can be a simple connection between the output variables of a functional module (e.g., Y_B^i of f_B^i in Agent *B*) to the input of a functional module in another agent. The conditional probability hypertables are in the form of $P(X_A^1 \mid Y_B^i, Y_C^j)$. In general, hyperlinks and hypertables may have complicated internal structures, reflecting the complex interaction patterns among agents.



Theoretically, this work will establish a systematic mathematical model for agent interaction under uncertainty. This model can support formal methods for a wide variety of inference tasks. It can be used to analyze and evaluate MAS behavior under uncertainty either by experiment through simulation or by formal analysis. It may serve as a basis for further theoretical development, including, for example, the models and methods for reaching consensus belief among agents, temporal models for agent interaction, and learning. Practically, this model may assist MAS developers by providing them the principles of and constraints on the way software agents are allowed to interact.

The Anthropomorphizing of Intelligent Agents

Tom Riley

NASA, Goddard Space Flight Center, Code 730.4, Greenbelt, Maryland 20771
John.T.Riley.1@gsfc.nasa.gov

Abstract. The name "agent" suggests that the program contains some quality of a human being. A programmer can intentionally create a virtual person to serve as the human/machine interface. The programmer should do this only if it will make the customer happy. Interfacing with other people does make people happy. Therefore, interfacing with virtual people may also make people happy if done well.

To do this requires an appropriate high level design for the agent. This design must include a controlling concept, a vocabulary, and a set of defining rules. If the design is to be practical, it must have well-defined limits. A wide selection of sources is currently available for developing suitable sets of rules. Object-oriented languages, like Java, and modular programming approaches provide an excellent environment for programming virtual people. The conceptual designs for two Web sites are developed.

1 Introduction

The very name “agent” conjures up an image of a person in a trench coat or a theatrical manager chomping on a cigar, but how far can we take this analogy? This paper discusses practical ways to design “virtual people” and explores the reasons for doing so. These ideas can be implemented at the very top of the agent design process.

The agent is basically a unit of software that provides a specific service for a customer. To qualify as an intelligent agent it must have some of the capabilities of artificial intelligence such as customizing the output to customer preferences it has learned from experience [1]. Such services can range from managing knowledge to playing soccer. An anthropomorphized agent or virtual person adds some quality of interaction with a human being to this service.

Agent software development work is being intently pursued by numerous teams, working mostly in commercial businesses but assisted by academic institutions and government agencies. For the most part, the teams, to maintain proprietary interests, are working privately and independently, so public academic discussion is critical to the process [2]. The team members tend to be strong on computer skills, but comparatively weak on human relations skills.

The success of an anthropomorphized agent, however, will turn on a team’s understanding of human needs, values, and relationships; their product must make their customers, the people who use their product, very happy if it is to succeed in the marketplace. This seems like a serious dilemma, but powerful new information and models are now available from a number of sources.

1.1 Software Tasks

The anthropomorphized agent development task can be broken down into a number of steps:

1. Define and limit the overall concept
2. Develop rules defining the state of being of the agent
3. Develop an image of the virtual person
4. Incorporate modular components such as data base management
5. Develop language handling

This paper concentrates on the first two steps and provides only a short discussion of the remaining three.

2 The Psychology of Happiness

Numerous business references [3] insist that we must make our customers happy and, first of all, our customers are human beings. Table 1 summarizes the results of numerous recent studies in psychology on what makes people happy [4]. It shows the main categories and subcategories of human activities that people report as making them happy.

Table 1 Summation of "The Psychology of Happiness" [4].

Main Category	**Sub-Categories**			
Social Relationships	*Marriage*	*Relatives*	*Friends*	*Neighbors*
Work	Intrinsic	Pay	*Co-workers*	*Supervisor*
Leisure	Skills	*Social Identity*	Relaxation	Other Worlds
Culture	Wealth	*Class*	Country	

The subcategories in italics are all concerned with human beings relating to other human beings. These make up the great bulk of the activities that people say make them happy. People like to interact with people. By extension, a well designed Web site, or even an operating system, that people experience as a person-to-person interaction should make them happy.

An anthropomorphized intelligent agent does not have to have the capabilities of a fully functioning human being. It does not have to pass Turing's Test [5] and fool someone into thinking it is human. It does not even have to have any common sense. It does have to provide the experience of interacting with a human being within the very limited boundaries of the application.

3 A Model of the Customer

Many recent studies provide breakthrough knowledge of how the human mind works [6], [7], [8], [9], [10]. This allows us to build a model of human mind function as shown in Table 2.

Table 2. Model of Mind Function

Free Will	Big Brain	This level can override all lower levels.
Socialization	Cultures	Here are the programs, skills, and biases humans derive from their society and environment.
Philosophy & Psychology	Rules for Being Human	The generalized rules affect all human beings.
Animal	Physical Processes	These physical brain structures are fundamental to survival.

This model is a work-in-progress: it is **not** presented as the "truth." An enormous number of new works are becoming available that will verify, refute, or refine it. The question here is whether this model can be used to build into our programs a quality of being human that the customer will truly appreciate.

Within this model, various human traits occur at various levels. Outside environmental factors trigger a trait and the human being responds. At any time a higher level may override a lower level. The overall result is like an autopilot which produces rapid responses to the environment in time of danger but which can be turned off at any time. The sum of these traits gives us the quality of being human. Possessing a small set of them can provide a limited quality of being human for anthropomorphized agents.

At the animal level are reactions such as Fight or Flight. In dangerous situations all human beings have a natural reaction, which is to save themselves. This is a trait that predates being human, yet it can be overridden by higher brain functions; when it is, we call it heroism.

At the Philosophy and Psychology level are reactions defining what it is to be a human being. These include being cynical or optimistic, persistent complaining, or just being who you are. The important ones at this level are the ones that all human beings share. They are common mental structures and part of being human but they too can be overridden.

The Socialization level includes the large amount of information we learn from interactions with other human beings around us. This is the often unconscious information humans rely on to get through the day, but computers do not inherently know any of it and it is quit difficult to teach it to them.

At the top of the model is Free Will which is a product of a large brain. We can simply override the automatic instructions coming from any lower level. Software agents do not need free will.

The task in designing an anthropomorphized agent, then is to define a small set of automatic reactions to expected inputs. These reactions must be human-like enough to support the purpose of the program. We are not trying to program a full set of human characteristics, only enough to make a Web site effective.

4 Sources of Input for the Model

There is such an enormous amount of new information available today that it simply swamps interested people. Here are some sources of materials, accessible to programmers, that can provide starting points for developing rule sets for anthropomorphized agents.

4.1 Psychology

Customers cannot tell us what they will want in the future. Only when we have something for them to experience can they truly react to it. We therefore propose to use general studies that define trends from instances of basic human qualities. Psychological testing has matured in the past few decades. Survey results are available that are accessible to programmers [4], [10] and fit in the Psychology category of our model.

These studies suggest that an anthropomorphized computer interface that occurred to users as a friend and that supported them in work and leisure activities, would make them very happy. This insight is the primary driver for anthropomorphizing agents.

For example, human beings like other human beings to be responsive to them [10]. An intelligent agent that occurs to the customer as truly responsive to the customer's requests should make the customer happier than one that miss responded.

Evolutionary Psychology is a new branch of psychology that looks at specific traits that have evolved in humans. Much of this work occurs as defining specific rules, and these rules can be programmed [8], [12].

4.2 Brain Physiology

The instrumentation is now becoming available to demonstrate the exact function of various sections of the human brain, the interconnection of the major pathways within the brain, and the location of chemical receptors [6]. These fit into the lowest level of our model. They have evolved by natural selection and are scientifically verifiable.

For example, most human beings, like many mammals, form long term pair bonds. These bonds are supported by specific structures in our brains [10] but require that the pair spend significant time together to nurture the bond so it will continue. This time together actually synchronizes the pair's internal clocks. Finding this time can be a problem in modern life. A Web site that celebrates and facilitates the couple's joint activities would thus provide a valued service for the customer.

Also an anthropomorphized intelligent agent that learns to match its timing to that of its customer would, theoretically, be well liked. This will require identifying the user's cycles and then applying the concepts of a phase-locked-loop.

4.3 Ontology

Ontology, the formal study of the nature of being, dates back to the ancient Greeks, yet it is important enough to be a primary source in the Philosophy & Psychology category of our model. It is already used Intelligent Agents [11], [12] but deserves a wider examination in the light of recent advances [13].

An excellent example of how defining being might apply to agent design can be seen in President Kennedy's "We shall go to the moon" speech. At that moment President Kennedy was being a great leader. People listened to him and got into action in large numbers. They stayed in action for a decade and achieved a historic milestone. Ontology studies those qualities exhibited by JFK at that moment that made him a great leader. What was the precise definition of his state of being? Was he following identifiable rules? Can others achieve great leadership through these rules?

This example can be understood as one of the strongest ontological concepts in the aerospace industry, "Buy-in." This is a common rule of human behavior that is widely recognized in high-tech industries. Buy-in has a number of distinct steps: becoming acquainted with an idea, visualizing oneself succeeding with the idea, publically supporting the idea, and getting into action. The period of action commonly lasts for years, or even decades. Buy-in is something that people do. It is not good or bad; it is simply a human trait.

An intelligent agent that included the concept of "buy-in" in its design might identify the stage in the process that the customer was currently in and provide services customized to that stage. It also might interact with the customer as an agent throughly bought into the Web site's service.

Academic studies of ontology, such as Jean-Paul Sartre's, "Being and Nothingness: A Phenomenological Essay on Ontology," are available but are not germane to the average programmer.

One popular version of ontology, however, is readily available to programmers and to the general public. The Landmark Education Corporation [15] is one of the largest non academic education companies in the world and offers courses based on ontology. Although limited in scope and considered proprietary by the company, these short courses can provide programmers with a clear understanding of how human beings respond to their environment with preset rules. Being able to identify some of these rules lets a person choose whether or not to take the prescribed actions. Identifying the rule gives that person power over how he or she will act within the domain of the rule.

4.4 Complexity Theory

Life lives on the edge of Chaos, an environment described by Complexity Theory [16]. A number of Complexity concepts are directly applicable to the Socialization category in our model as most social and economic systems qualify technically as complex systems. Several concepts are shown in Table 3.

Table 3. Concepts from Complexity Theory that are Relevant to the Design of Anthropomorphizing of Intelligent Agents

Emergent Results	The whole can be more powerful than the sum of the parts. Such results are not predictable from the sub-components alone: such results cannot be reverse-engineered. The Mandelbrot set [17] is a clear example.
Interconnectivity	Necessary for complexity, critical to learning, and it creates the principal value of the Internet.
Self-Organizing	Complex systems appear to organize themselves.
Attractors	Concepts can attract actions; in this case the consumer's dollar attracts all the computer programmers in the race and thereby drives the action.

4.5 Anthropology

Religion, culture, myths, and social norms provide rules by which people live and make up the Socialization category in our model. Anthropology has made great strides in recording the knowledge of indigenous peoples [18]. This opens up the possibility of virtual people who are a celebration of a group and not a stereotype of one.

Dr. Joseph Campbell was one of the great teachers of the twentieth century. His work [19] centered on the myths of indigenous people and their value to modern cultures. The outstanding quality of his writings and lectures makes this information accessible to programers. Myths are given in language and language is a key step in getting from concept to real software anthropomorphized agents.

4.6 Business Practices

The literature on good professional and business practices can provide specific rules for the efficient and ethical use of agents. These too fall under Socialization in our model. Rules to make your customers happy and operate a business with integrity should be particularly valued. Popular authors in this field could prove particularly attractive to the aggregate of business computer users.

5 Web Site Design Examples

The following three design examples demonstrate the concept of this paper. They feature a NASA educational outreach task, a "Clearing for" mountain climbing, and an historic example.

5.1 GSFC Educational Outreach Example

The Congress of the United States has mandated NASA to reach out to American students and provide high-tech learning experiences that draw on NASA projects. Consider a possible case: how anthropomorphized intelligent agents could be used for education outreach.

NASA has a major project called "Earth Observation System" (EOS)[20]. It will operate a constellation of satellites producing terabits of data about Earth's environment for more than a decade. The flagship satellite, Terra, was successfully launched in December 1999. EOS will produce a prodigious amount of data and is a resource of immense value.

NASA's education outreach effort that will take advantage of this resource. It will have the characteristics shown in Table 4. We could organize this educational effort as a series of windows, but we here choose to increase the happiness of our customers, students, and teachers by including three virtual People. They are the Teacher, the Student, and the Coach.

The virtual Teacher has the straightforward programmed activities: present the lessons, answer questions, and track the progress of student team members. The virtual Teacher will keep track of progress through the lessons and fit the lesson to the teams progress. Questions requiring clarification of the lesson will be answered. Questions that are simple restatements of student homework intended to fool the virtual Teacher into answering the homework will not be answered. Questions of the Frequently-Asked-Questions (FAQ) type will be answered if time allows. Intelligent functions include keeping track of the progress of each team and each student in the team, and choosing the right material for the team's level.

Table 4. Requirements for NASA Educational Outreach Example

Lesson Plans	A series of lessons complying with formal state and professional teaching guidelines, and educational standards.
Web Site	The presentation will be computer based. This promotes an interactive environment and access to the data.
Students	Small teams of two to four high school students. The size of the team is set by the availability of computer equipment but it is not an individual activity.

The virtual Student is a subordinate member of the student team. This program element will fetch data requested by the team both from the EOS archive and from other sites on the Web. It will also be ready to play team building games. For many of the students who are not interested in a career in science the team building activities will be the most important part of the lesson.

The virtual Coach has the most complicated ontology but, fortunately, substantial amounts of literature on good coaching are available. The virtual Coach is basically a Fault Detection and Correction program that monitors a student's activities. If the student wastes time or complains about the amount and difficulty of the work, the virtual Coach will respond with appropriate actions ranging from pep talks to dressing down. When a student is stuck, the virtual Coach must change the student's state of being. The virtual Coach wears a baseball cap and whistle. The virtual Coach is very easy for the students to identify and relate to.

In a typical session the virtual Teacher presents new material and lays out an assignment. The virtual Student assists the team in finding the materials needed for the assignment. The virtual Teacher fields questions during the work session, and the virtual Coach addresses complaints and excuses while providing inspiration and team building. Success may be rewarded with play time in which the virtual Student participates as a member of the student game team.

5.2 Clearing for – Mountain Climbing Example

Web sites using the concepts of anthropomorphized intelligent agents do not necessarily have to show a representation of a person. Animations that do not contain most of the content of Web page actually detract the reader from finding information [21]. Instead, consider when you read a good book, you let the author take over the voice in your head (called your Subvocal Linearization). This is a restful and enjoyable experience, but requires quality writing.

For example, consider a hypothetical Web site that sells mountain climbing equipment. With a conventional design it would include a database of items for sale and shopping basket software to buy them. With an ontological design, the Web Site is a conversation [13] that takes the form of a "Clearing for" and a "Celebration of" the mountain climbing experience, as shown in Table 5.

The design model for “Clearing for” is a clearing in a forest. A person who has been walking through a dark and scary wood breaks into a clearing with sunlight and good water. The effect is reassuring and frees the person to think about what is present in the clearing. In short, we will produce a pleasurable state of being for the customer by providing an environment with integrity that invites participation on a topic the customer loves.

There is no representation of a person in this Web site, but the site itself has the feel of a conservation with a friendly salesperson who has loads of expertise about mountain climbing and the equipment for sale. It represents a timeless clearing for the customer to talk about the sport and gives access to all the data the customer needs. The site relies on a well-written trail guide, as well as technical information regarding the company's products and current stock. The site is primarily database oriented.

The customer can buy equipment, or simply stop to talk. The site then invites the customer to remember, or imagine for the first time, the states of being involved in mountain climbing. First comes the planning, with all the anticipation; then the climb, with great concentration on the technical skills; then the time at the top, with its exhilaration; and finally, the trip down. Each step is supported with text descriptions and multimedia materials. The quality of the writing is the key to keeping the customer inside the experience.

Table 5. Key Ontology Concepts in the Mountain Climbing Web Site.

Clearing for	The Web site creates a safe space in which something can happen.
Celebration of	The author of the Web site has a clear love and deep experience of the topic.
Invitation	The customer is invited to participate. There are no consequences for not doing so and no coercion to buy something.
Ontology and Language	Ontology is a function of language. Graphics can provide support, but without language there is no definable state of being.

This design requires that the Web site owner purchase Web rights to a well-written mountain climbing guide. A very small amount of the company's computer time is required for the additional data to pass to the customer's computer. The bulk of the equipment time is on the customer's equipment at **no** cost to the Web site owner. If the Web site makes its customers happy then the resulting word-of-mouth advertizing will far exceed in value the extra implementation costs.

5.3 Historic Example – ELIZA

In 1966 Joseph Weizenbaum at Massachusetts Institute of Technology wrote a deceptively simple program that mimics the talking-cure counseling style used in psychoanalysis. The program, called ELIZA, was intended as a simple exercise moving toward meeting the Turning test [5]. The unexpected effectiveness of the program came as a big surprise and raised integrity issues. A Java version is currently available over the Web [22].

The program accepts text typed in from a terminal and it outputs text in reply. The replies are questions eliciting additional information on the data in the input stream.

The person at the terminal is strongly inclined to keep the exchange going, and many report the exchange was meaningful for them. The program in no way comprehends the exchange and cannot possibly provide the person with any help on the problems discussed. Controversy arose when many respondents were observed using the program as if it were a real counselor.

Thus ELIZA is a powerful example of a simple anthropomorphized program that makes the user happy. The program searches the input stream for key words. Simple rules are used to convert key words into a question or a statement like "Tell me more about ...". This produces enough of a feel of talking to a fellow human being, one who is responsive to the communications, that the user often reveals deep anxieties and emotions.

This result can be understood in terms of simple human traits. People like to be paid attention to and have that attention be responsive to them [10]. ELIZA's lesson for us is that uncomplicated anthropomorphized programs can create powerful responses. If we choose to use this approach, we must design the agent with care and consideration.

6 Programming

Our programming teams must work their way from vague concepts and obtuse jargon to hard computer code. To do this they need to design an ontology (state of being) with the following elements:

6.1 Top Level Design – A Defining Concept

The top level design defines the area of activity, communicates the most critical concepts, and provides limits. We are not trying to program a general purpose human being, so defining the limits of the agent is critical.

An anthropomorphized agent could be used as the human/machine interface for a number of activities, including both operating systems and Web sites. Operating systems require an enormous investment to develop and have huge installed bases, while Web sites do not. Individual Web sites are therefore most likely to be the first applications.

Most applications will have one clearly defined purpose, such as sales [Section 5.2] or instruction [Section 5.1]. This purpose must be clear to the customer. The application will always have a second purpose: to make the customer happy. This purpose must be clear and sincere, too.

The application will probably have a visible anthropomorphized image. This image needs to be well planned and tested. The principles in this paper can also be applied without an actual image [Section 5.2].

The source of the rules must be chosen and, if necessary, intellectual property rights secured. This will typically also apply to any databases used.

This whole design effort is about man/machine interfacing. The exact physical means of this interface is critical. It will usually be a text input stream originating either from a keyboard or from a voice recognition system. Other inputs such as point-and-click may be included. It will be necessary for the program designers to have a clear understanding of how the customer likes to use all the available inputs.

6.2 Elements to Code

General design concepts, as shown in Table 6, must be developed into forms that can be programmed. The anthropomorphized concepts are relatively easy for programming teams to visualize and execute as, since the programmers too are people. These concepts are also exactly what is needed for object-oriented programing.

Precise definitions of jargon must reduce all the way down to plain language. Most current examples of IA ontologies are primarily vocabularies. This vocabulary will be application specific. In the teaching example [Section 5.1], the virtual Coach must handle complaints from the students. A detailed vocabulary of complaining as used by high school students is needed.

These types of rules are easily coded in "if-then", "is-a", or "has-a" forms in object-oriented languages such as Java. They may also be related to key words from

Table 6. High-level design consideration for anthropomorphized intelligent agents.

Vocabulary	Ontology is a function of language.
Rules	To turn ideas into programs, the vocabulary must be supported by specific methods or rules.
Data Access	Data access in needed for practical applications. Formal data bases will be common.
Image	The application may include a window with an image of a person or anthropomorphized animal or thing.

the vocabulary, grammatical structures (such as a question), and ontological concepts (such as a complaint).

Agent images may be generated by applying outline makeup to a real person and applying white paper reference dots to the person's face [23]. The person can then act out facial expression before a digital camera. If the image is refreshed more than 15 times a second, then a feeling of the image being present and responsive is possible.

6.3 Modular Code

We must learn to crawl before we can fly. The widespread use of anthropomorphized agents in Web sites will lead to success in more demanding applications and perhaps even to passing Turing's test [5]. This path is strongly supported by the trends in modern software design as shown in Table 7.

When key modules are available, the creation of anthropomorphized agents will be primarily a high-level design effort and, quite possibly, a new art form.

Table 7. Currently available software trends that support anthropomorphized intelligent agents.

Java	This modern object-oriented language is optimized for the Web.
Java Beans	These will allow the development of modular units for all the common functions that can be used as objects.
Data Base Handler	This will probably be a very common function for an agent [24].
Input Text Handler	Applications will start primarily with a keyboard input but should graduate to voice recognition very soon.

7 Conclusion

Commercial programming teams should anthropomorphize their intelligent agents if, and only if, doing so will make their customers happy. Many sources of concepts and rules for such designs are available and accessible. This effort can add a human quality to an intelligent agent, making that agent easy for the customer to enjoy using.

References

[1] Joseph P. Bigus, Jennifer Bigus, "Constructing Intelligent Agents with Java", Wiley Computer Publishing, New York (1997). Provides coded examples of intelligent agents that search, represent knowledge, reason with rules, and learn.

[2] Tracy Kidder, “The Soul of a New Machine”, Little, Brown and Company, Boston (1981). Still the best description of the process of bringing a high-technology product to market.

[3] D.H. Stamatis, “Total Quality Service”, St. Lucie Press, Delay Beach, Fl, (1996).

[4] Michael Argyle, “The Psychology of Happiness”, Routledge, London (1987). Summarizes forty psychology studies over an eleven year period on what makes people happy.

[5] A. M. Turing, “Computing Machines and Intelligence” 1950 in "Mind Design II, Philosophy, Psychology, Artificial Intelligence” edited by John Haugeland, The MIT Press, Cambridge, Massachusetts (1997), pp 29. A defining article for fifty years.

[6] Antonio R. Damasio, “How the Brain Creates Mind”, Scientific American, December 1999, pp 112-117

[7] Steven Pinker, “How the Mind Works”, W. W. Norton & Company, New York, (1999)

[8] Robert Wright, “The Moral Animal, Evolutionary Psychology and Everyday Life”, Vintage Books, New York (1994)

[9] Matt Ridley, “The Origins of Virtue, Human Instincts and the Evolution of Cooperation” Penguin Books, New York (1997)

[10] Thomas Lewis, M.D., Fari Amini, M.D., and Richard Lannon, M.D., “A General Theory of Love”, Random House, New York, (2000)

[11] N. Guarino, Editor, “Formal Ontology in Information Systems: Proceedings of the First International Conference June 6-8, 1998, Trento, Italy”, I. O. S. Press, (1998).

[12] What is an Ontology? Tom Gruber.
http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[13] Terry Winogard and Fernando Flores, “Understanding Computers and Cognition”, Addison-Wesley, Reading, Massachusetts (1997)

[14] Henry Gee, “In Search of Deep Time”, The Free Press, New York (1999).

[15] Landmark Education Corporation. February 2000. http://www.landmark-education.com

[16] M. Michel Waldrop, “Complexity, The Emerging Science at the Edge of Order and Chaos”, Simon & Shuster, New York, (1992)

[17] Explore the Mandelbrot Set, Fractal Explore Kit. Tom Harrington
http://home.rmi.net/~tph/factalkit/mandel.html

[18] Jared Diamond, “Guns, Germs, and Steel, The Fates of Human Societies”, W. W. Norton & Company, New York (1997). This book is written as a disproof of racism, but in the process the author lays out very clearly why human beings are going into space.

[19] Joseph Campbell, “Myths to Live By”, Bantam Books, New York (1972). An example of the lectures of a great teacher.

[20] Terra, The EOS Flagship. NASA. Feb. 2000. http://terra.nasa.gov/

[21] Jared M. Spool, Tara Scanlon, Carolyn Snyder, Will Schroder, Terri DeAngelo, “Web Site Usability: A Designer’s Guide”, Morgan Kaufmann Publishers, Inc. San Francisco (1999).

[22] ELIZA as a JAVA Applet. Robert C. Goerlich, February 3, 1997
http://philly.cyberloft.com/bogoerlic/eliza.htm

[23] F. Lavagetto, “Multimedia Telephone for Hearing-Impaired People” in “Fusion of Neural Networks, Fuzzy Sets, and Genetic Algorithms” edited by Lakhmi C. Jain, N. M. Martin, RCR Press, Roca Raton (1999), pp 257–294. This article provides details on how a facial image can be generated with enough fidelity to allow lip reading. I believe the same level will allow a customer to accept an image as humanly interactive.

[24] Jesse Feiler, “Database-Driven Web Sites”, Morgan Kaufmann Publishers, Inc., San Francisco (1998).

Controlling Multiple Satellite Constellations Using the TEAMAgent System

Derek M. Surka[1], Mark E. Campbell[2], and Thomas P. Schetter[2]

[1]Princeton Satellite Systems, 33 Witherspoon St., Princeton, NJ 08542
dmsurka@psatellite.com
[2]University of Washington, Department of Aeronautics and Astronautics, Box 352400, Seattle, WA 98195-2400
mcamp@aa.washington.edu, thomas_schetter@hotmail.com

1 Introduction

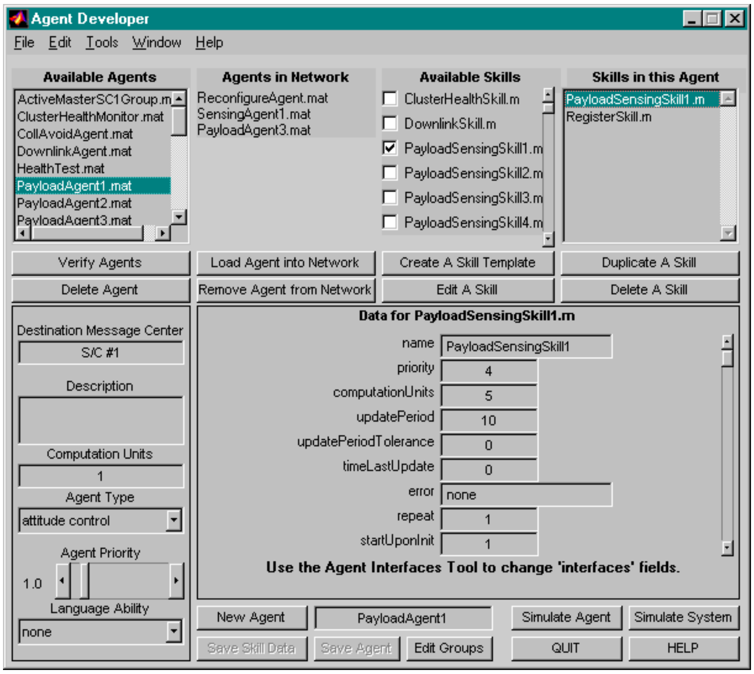
There is an increasing desire in many organizations, including NASA, to use constellations or fleets of autonomous spacecraft working together to accomplish complex mission objectives. Coordinating the activities of all the satellites in a constellation is not a trivial task, however.

The TEAMAgent system is being developed under a Phase II SBIR contract from the Surveillance and Control Division of the Air Force Research Laboratory's Space Vehicles Directorate to address this issue. The TEAMAgent system is an extension of the ObjectAgent software architecture to constellations of multiple spacecraft. The required spacecraft functions for the multiple spacecraft missions have been identified and the use of software agents and multi-agent based organizations to satisfy these functions have been demonstrated. TEAMAgent is presently being ported from Matlab to C++ for implementation on a real-time system.

2 The ObjectAgent Software Architecture

ObjectAgent is a new software architecture for real-time systems which require complex autonomous decision making as part of their normal operations. Agents are used to implement all of the software functionality and communicate through messages using a simplified natural language, which enables end-users to easily understand and command them. Decision-making and fault detection and recovery capabilities are built-in at all levels of the software, which alleviates the need for extremely intelligent high-level agents. Furthermore, agents can be dynamically loaded at any time, which simplifies the process of updating flight software and removes the complexity associated with software patches.

The ObjectAgent software package provides a graphical user interface (GUI) based development environment for the design and simulation of multi-agent systems. This design environment simplifies the agent creation process and provides a common interface to a number of advanced control and estimation techniques. The Agent Developer GUI is shown in Figure 1.



Available Agents: ActiveMasterSC1Group.m, ClusterHealthMonitor.mat, CollAvoidAgent.mat, DownlinkAgent.mat, HealthTest.mat, PayloadAgent1.mat, PayloadAgent2.mat, PayloadAgent3.mat

Agents in Network: ReconfigureAgent.mat, SensingAgent1.mat, PayloadAgent3.mat

Available Skills: ☐ ClusterHealthSkill.m, ☐ DownlinkSkill.m, ☑ PayloadSensingSkill1.m, ☐ PayloadSensingSkill2.m, ☐ PayloadSensingSkill3.m, ☐ PayloadSensingSkill4.m

Skills in this Agent: PayloadSensingSkill1.m, RegisterSkill.m

Destination Message Center: S/C #1

Description:

Computation Units: 1

Agent Type: attitude control

Agent Priority: 1.0

Language Ability: none

PayloadAgent1

Data for PayloadSensingSkill1.m

name	PayloadSensingSkill1
priority	4
computationUnits	5
updatePeriod	10
updatePeriodTolerance	0
timeLastUpdate	0
error	none
repeat	1
startUponInit	1

Use the Agent Interfaces Tool to change 'interfaces' fields.

Fig. 1. The Agent Developer GUI allows users to create agents without writing a lot of code.

3 The TEAMAgent System

The TEAMAgent system builds upon ObjectAgent and enables agent-based multi-satellite systems to fulfill complex mission objectives by autonomously making high- and low-level decisions based on the information available to any and/or all agents in the satellite system. This is achieved through the use of hierarchical communication and decision-making paths among the agents that can be autonomously reconfigured as new agents are added and old agents are modified or removed. GUIs enable the satellite end users to initially configure the TEAMAgent system and monitor the effectiveness of the decisions made by the agents.

Simulations of multi-agent systems for multiple satellites have been developed using TEAMAgent, such as collision avoidance maneuver and reconfiguration for a four-satellite constellation. Agents are used to monitor for collisions, reconfigure the fleet, optimize fuel usage across the cluster during reconfiguration, and develop a fuel optimal maneuver for reconfiguration. These simulations have demonstrated the feasibility of using TEAMAgent to control multiple satellite constellations and TEAMAgent is presently being ported to C++ for implementation on a real-time testbed and satellite system.

Operations Assistants for Johnson Space Center's Mission Control Center

Susan E. Torney

Mail Code DV
Lyndon B. Johnson Space Center
National Aeronautics and Space Administration
2101 NASA Road 1
Houston, TX 77058-3696
susan.e.torney1@jsc.nasa.gov

The Mission Operations Directorate (MOD) at JSC is responsible for the planning and conduct of human space flight missions. MOD is being challenged with developing and sustaining new operations capabilities to support the growth in existing programs and to enable the advent of the human exploration missions. Mission operations are becoming increasingly distributed due to the growth in international participation and the expanded involvement of the scientific, academic, and industrial communities. Flight controllers are challenged by new roles and complex operational scenarios such as multi-vehicle control, long distances, and extended duration operations. Support for long distance exploration missions will require a shift in real-time control from the ground to onboard the spacecraft due to the inherent communications delays. In addition, MOD is continuously seeking to improve its processes to accomplish missions at higher levels of safety, mission success, and effectiveness. Automation is being considered as an enabling technology to meet the aforementioned challenges. The synergistic combination of flight controllers and intelligent software providing the function of Operations Assistants is being pursued as the key implementation of this technology in the Mission Control Center (MCC).

Although successfully implemented in a variety of unmanned areas such as satellites and robots, automation technology cannot be transferred directly to the MCC due to the unique challenges of manned space flight (see following section). Therefore, a team of experts from across NASA and academia is working to apply and extend state-of-the-art automation technology to manned space operations.

Formal methods are necessary to ensure the stringent levels of safety and reliability required for the manned space program. Formal methods are also key to successful development, maintenance, and certification processes. To date, Data Flow Diagrams and Petri Nets have been utilized in the analysis of the duties of one MCC position.

Please feel free to contact the author about (1) the current MCC project, (2) formal methods, (3) the use of Petri Nets, and (4) any other related questions.

Challenges to Implementing Intelligent Agents in Manned Space Operations

- SAFETY – Lives and national assets are at stake
- TEAMWORK – Operations Assistants must interact with the Flight Controllers in such a way as to function as part of the flight control team. They will essentially be software-based "back-room flight controllers."

- RELIABILITY – Flight Controllers must be able to trust the software (i.e. understand how it works and trust that it will make the right decisions every time). The software must be certified for use in the Mission Control Center.
- EASE OF DEVELOPMENT & MAINTENANCE – Flight Controllers must be able to easily update and modify the software when vehicle hardware or operations concepts (flight rules, procedures, etc.) change. Flight Controllers must be able to customize the software for their disciplines.
- CHALLENGES OF FUTURE PROGRAMS – Exploration missions will have added challenges due to the long distances and long durations. Many current MCC functions will move onboard the vehicle.

** *Formal approaches are mandatory for success in all of these areas*

Mission Control Center (MCC) Operations Assistants

- OBJECTIVES:
  1. To develop and implement Operations Assistants that will function as members of the flight control team and assist flight controllers in
     - maintaining mission cognizance
     - performing duties such as
       - Data analysis
       - Failure identification
       - Off-nominal situation response
       - Planning
       - System/process management
  2. To mature the technology for future programs
- APPROACH: Implement intelligent agent technology by focusing on these four areas: (1) architecture, (2) intelligent agents, (3) knowledge capture processes and applications, and (4) advanced human interfaces

Initial Development Phase: Logging Assistant

- CURRENT CAPABILITIES:

– Maintains flight controller logs (their notes about mission activities) in a data base
– Automates the handover report process

- THIS ENABLES EXTENSIVE FUTURE CAPABILITIES:

– The use of advanced data base features such as (1) intelligent searches, (2) tracking of open issues, and (3) item linkages
– Connections to vehicle data (telemetry) and information resources (flight rules, procedures, etc.) for (1) automated logging and (2) automated presentations to flight controllers for routine operations and failure responses
– Connections with existing MCC software tools
– Connections with intelligent agents

A Real Time Object-Oriented Rational Agent Development System[1]

Leonard P. Wesley

Computer, Information and Systems Engineering Department
College of Engineering
San Jose State University
One Washington Square
San Jose, CA 95192-0180
408-924-3928 (Voice)
lwesley@intellexus.com

Abstract. A real-time object-oriented agent development system (ROADS) is being developed and used to build embedded distributed rational agents in a manner that is difficult or not possible to build with existing agent development environments(ADEs). Some ROADS innovations include the use of a theory of objects as a foundation on which; (1)different agent development languages can be defined and used within a single ADE; (2)computational models of beliefs desires and intentions can be implemented (3)possible world semantics are readily supported, (4)reactivity and responsiveness are under direct dynamic control of the agent applications; and (5)a formal foundation to model cooperative multi-agent applications is provided. ROADS has been successfully used in several complex applications such as distance learning, high-level telecommunications network management, and eventing systems.

1 Introduction

The use of automated agent-based approaches and technologies continues to expand into increasingly complex and diverse task domains. Some agents must meet stringent real-time requirements, others must be highly mobile, some might require sophisticated deliberative capabilities to carry out assigned tasks, and others might need to coordinate their activities with other distributed agents. Given the typical level of effort and cost with developing agents, it is very important to select the most appropriate agent development environment (ADE) within which to build agent-based applications. Often it is too costly to simply rebuild an agent in a different ADE if the current environment is not suitable, or to use a different ADE for comparative purposes. While no single ADE is the "end-all" to agent development activities, it would be useful to have an ADE that is capable of providing the features of several

[1] The work described in this article was supported, in part, by NSF CISERI equipment grant No. EIA-9818335, San Jose State University, CISE Department, and Intellex, 5932 Killarney Circle, San Jose, CA 95138

alternative ADEs and to support translating between different ADE languages, and so forth.

While no such ADE currently exists, the number and type of ADEs continues to evolve and expand. Some well know examples are described in Huhns (1997) [1]. Examples include the rule-based Agent Building Environment (ABE) from IBM that is written in C++ and Java. Stanford University's Java Agent Template (JAT) supports communication over LANs via KQML that was defined under DARPA's sponsorship. The Java Expert System Shell (JESS) allows the development of single standalone agents and is essentially a rewrite of CLIPS in Java. Stanford Research Institutes (SRIs) CYPRES and Procedural Reasoning System (PRS) is an agent development and planning environment that is written in LISP, Wilkins (1994). SRI's PRS has been successfully applied in a large number of applications to meet distributed real-time embedded system requirements. Within PRS and CYPRES, agents are written using the ACT language, Wesley (1991, 1993a, 1993b) [2, 3, 4]. SRI's Open Agent Architecture (OAA) system supports the development of heterogeneous agents that communicate using a logic-based declarative InterAgent Communication Language that runs on top of CORBA. One of the OAA agents distributes tasks to other agents. The OAA is advertised as an environment where agents can be created using multiple programming languages. However, one must select from a library of OAA implementations that provide the desired language. If you want to change the language, you select a different OAA library. The Australian Artificial Intelligence Institute's Distributed Multi-Agent Reasoning System (dMARS) is a C++-based ADE that has origins in PRS, similarly Ingrand's (1992) C-PRS system is a C implementation and a derivative of PRS [5, 6]. Sutton's (1992) Dyna architecture allows one to define agents that can learn from environmental rewards and then use these agents for planning purposes[7]. Corkill's (1991) well known blackboard architectures provide a means for agents to communicate and coordinate their activities. To date none of these admittedly useful and successful systems appear adequate to meet some of today's important agent-based challenges.

ADEs that are suitable for developing autonomous agents must be capable of meeting several important requirements. These include providing a means to rapidly react and respond to environmental events. They must also support the development of agents that carry out top-down and bottom-up deliberative activities. ADEs must also provide a development language that is more abstract than traditional C/C++ or Java type languages. Rao and Georgeff (1991) provide compelling arguments for ADEs that provide language primitives to manipulate notions such as "beliefs," "desires," and "intentions," commonly referred to as BDI architectures, Bratman (1987) [8]. It is more desirable to build agents using statements like "*Form the intention to execute some action if we believe some condition about the state of our environment* " than using C/C++ *IF-THEN* type language constructs. It must also be anticipated that ADEs and agents built therefrom will have needs to scale up to larger and more complex problems. To date the blending of object-oriented and system engineering techniques to help address system life-cycle issues have not received adequate attention.

2 The ROADS Approach

The foundation of our approach to building agents within ROADS is illustrated in Figure 1. All ROADS application agents have the same kernel which consists of three asynchronously running processes named REACT, RESPONSE, and EXECUTE. The primitive entity that ROADS manipulates is an "object." Everything to the ROADS kernel is an object that is defined in Figure 2. Objects have a name, id, priority, data fields and one or more methods that manipulate the data. The REACT process (#2 in Figure 1) receives information about the external world(#1 in Figure 1) in the form of objects that are ASCII text strings. Received objects are then passed(#3) to the RESPONSE process(#4) that determines how the agent should handle the object by executing, as an asynchronous thread(#5), the method specified in the response slot of the object. At times no further action is required and the object is discarded. Other times one or more specific actions are warranted. This is accomplished by the response method passing(#6) an object to the EXECUTE(#7) process which will execute, as asynchronous threads(#8), one or methods specified in the execute slot of the passed object. EXECUTE threads usually do most of an agent's work. They interact with the external world via tcp/ip socket communication(#10). A memory resident database (#9) along with access functions are available to store and retrieve application specific information. In addition, EXECUTE threads can pass(#11) new objects to the REACT process which in turn get processed as just described. Agent applications can be built by constructing the desired objects and associated methods.

How can ROADS meet the previously stated requirements? Because REACT, RESPOND, and EXECUTE are asynchronous processes an agent can "react" to incoming events while "responding," such as executing EXECUTE functions(#8 Figure 1), to others. This is particularly evident on platforms having three or more CPUs, where the latency between receiving and object and responding to it was observed to be on the order or milliseconds on a Pentium-233Mhz platform running Solaris. Furthermore, ROADS provides a special function that an object can specify to dynamically control an agent's relative degree of reactivity and responsiveness. The `object_data` field of the object specifies how many objects REACT will queue before passing them to RESPOND. Queuing more objects before passing them will increase reactivity. Queuing fewer objects before passing them to RESPOND will increase responsiveness. Having this capability under agent control is important when downloading material that might arrive from different locations at different rates, and simultaneously processing user requests or displaying previously retrieved information. How ROADS meets ADE language development requirements is described in the following section.

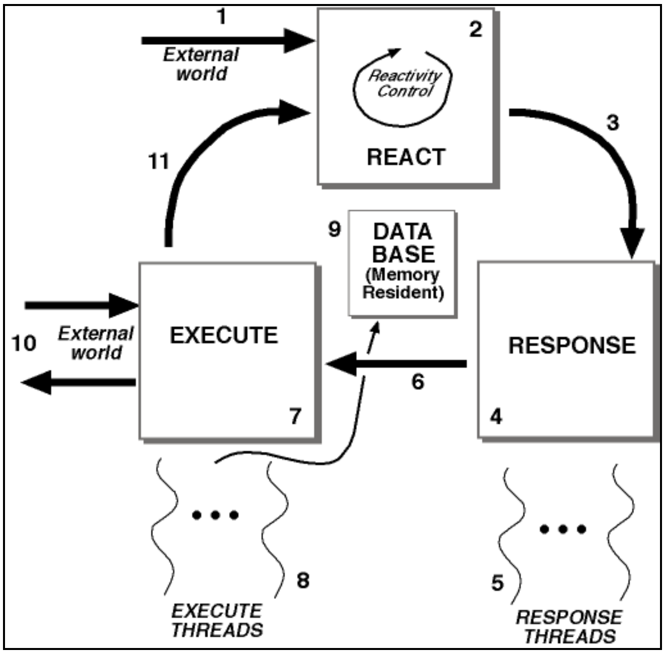


Fig. 1. Illustration of ROADS kernel

```
typedef struct {
   long prty;                  /* priority */
   short msg_len;
   char obj_name[MAXLENGTH];   /* name of */
   short mid;                  /* unique id */
   char response[MAXLENGTH];   /* response fun name */
   char execute[MAXFUNCTIONS]; /* execute fun(s) name */
   pid_t pid;
   short print_type;            /* TESTING ONLY */
   ObjDataType object_data;     /* appl data field */
#ifdef PMONITOR
   double rec_time;     /*   for response */
   double exec_time;    /* for execute ONLY */
   double on_qtime;     /*   time on queues */
#endif
}OBJECT;
```

Fig. 2. Structure of a ROADS object.

2.1 A Theory of Objects

Given the general nature of objects, its clear that object-data and associated methods to manipulate this data can be specified to carry out arbitrarily complex tasks. This object-based foundation of ROADS serves as the common language on which other agent-based languages can be defined and compared. This is in contrast to many other ADEs that provide just one agent development language, or requires the user to pre-load a language library.

Formally, we define $\varsigma(x)b$ as a method with parameter x and body b. We also let

$$[l_1 = \varsigma(x_1)b_1 l_1, ..., l_n = \varsigma(x_n)b_n] \qquad (1)$$

represent an object with n methods labeled $l_1, ..., l_n$, and where $o.l$ represents a method l of object o, and $o.l \Leftarrow \varsigma(x)b$ represents the update of the method of object o with method $\varsigma(x)b$. While class-based languages appear fully formed in systems such as Simula, object languages are intended to be simpler and more flexible than traditional class-based languages, Abadi (1996) [9]. In a minimal form, object-based languages only support the notion of objects and dynamic dispatch. When objects are typed, notions of subtyping, subsumption (i.e., abstraction), and formalizations of concepts derived therefrom can then be supported.

Why make objects the foundation of an ADE? Our intention is to bring to bear the theory, practice, and tools of objects to allow one to define, predict, and prove certain properties about agents. Users need answers to questions like, 'Can I accomplish <x> with agent language <y> while meeting constraints <z>?', 'Can <a> be accomplished more efficiently using language <b> than using language <c>?", and 'Can some property or condition <d> be proven using language <e>?" The object calculus (ξ-calculus) of interest here consists of a minimal set of semantic constructs and computational rules. The handling and proof of properties about objects and the ability to meet constraints can be expressed as an application of a sequence of reduction steps as specified by a set of reduction and equivalence relations on terms in the ξ-calculus. The execution of a ξ-calculus term such as a method of an object is also expressed as a sequence of reduction steps. The execution of method $o.l$ reduces to a ξ-calculus term as specified by a set of reduction relations that prescribe how properties, parameter values, and so forth can change. One builds the desired logic-based, IF-THEN, BDI and so forth agent building languages using the ξ-calculus. For example, an IF-THEN rule might be implemented by placing the antecedent to check and consequence to assert in the `user_data` field of an object. and providing a RESPONSE function that evaluates the antecedent and if satisfied passes an object that contains an EXECUTE function to assert the consequence as a thread in the EXECUTE process.

Comparing aspects of agents written in if-then languages with agents written in, say a BDI language, is accomplished, in part, by analyzing and proving properties of the underlying ξ-calculus on which the respective languages are written. Space precludes describing how the ξ-calculus and reduction rules can be used to assess whether an object's methods will execute within given time constraints or if an algorithm meets soundness and completeness constraints.

2.2 Possible World Semantics

A preliminary set of ROADS objects in conjunction with its DB are available to support possible world semantics where facts, beliefs and so forth can be indexed by time or keyword that is associated with application specific hypothetical worlds containing a logically consistent set of hypothesis, intentions, or desires.

2.3 System Life-Cycle Issues

ROADS objects are designed to be inherited and or "overloaded" in much the same manner as C++ classes and methods but in a way that is transparent to a user or higher-level language. This allows applications to be developed by building upon previously developed objects and methods. It is well known that these C++ notions facilitate code/object reuse, maintaining code/objects, and extending applications in less time than is possible without such facilities.

References

[1] Huhns, Michael and Munindar P. Singh, (1997) *Readings in Agents*, Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[2] Wesley, Leonard P., (June 1991) "Application of PRS to Network Management Systems," Final Project Report for an International Japanese company, Tokyo, Japan.

[3] Wesley, Leonard P, &Hashimoto, Kazio , & Tohuro Asami, (March 23, 1993a) "International Telephone Network Management as Distributed Problem Solving: Investigation of a Practical Strategy and Solution Tradeoffs," In Proceedings. IEICE (Institute of Electronics, Information and Communication Engineers), Vol 92, No. 529 (AI92 102-109), pp. 17-24, Tokyo, Japan,.

[4] Wesley, Leonard, & Janet D. Lee, (March 31, 1993b) "Evaluation of C-PRS," A client private Final Project Report for an International Japanese company, Tokyo, Japan.

[5] Rao, Anand S., & Georgeff, Michael P., (1991) "Modeling Rational Agents within a BDI-architecture," In J. Allen, R. Fikes, and E. Sandewall, Eds., *Proc. Of the Second International Conference on Principles of Knowledge Representation and Reasoning.,* Morgan Kaufmann Publishers, San Mateo, CA.

[6] Ingrand, Felix F., M.P. Georgeff, and A.S. Rao, (1992) "An Architecture For Real-Time Reasoning and System Control," In *IEEE Expert,* 7(6).

[7] Sutton, Richard, (1991) "Dyna: An Integrated Architecture For Learning, Planning and Reacting," Working Notes of the 1991 AAAI Spring Symposium on Integrated Intelligent Architectures and SIGART Bulletin 2, pp. 160-163.

[8] Bratman, M.E., (1987) "Intentions, Plans, and Practical Reasoning," Harvard University Press, Cambridge, MA.

[9] Abadi, Martin and Luca Cardelli, (1996) "A Theory of Objects," David Gries and Fred Schneider Editors, Springer Verlag, New York City, New York, USA, Heidelberg, and Berlin Germany, Berlin.

Panel Discussion: "Empirical Versus Formal Methods"

1 Moderator's Summary of the Panel Discussion

Diana Gordon
AI Center, Naval Research Laboratory (Code 5514)
Washington D.C. 20375

The panel on Empirical versus Formal Methods was highly thought-provoking. The panel began with 10-minute presentations by the panel members. The first speaker was Doug Smith from Kestrel Institute. The main thrust of Smith's presentation was that formal methods enable run-time matching of agent services and requirements. In particular, if agent services and requirements are formally specified, then it is possible to automate the matchmaking process. Smith's presentation was followed by Henry Hexmoor, from the University of North Dakota. Hexmoor emphasized the need for a synergistic relationship between empirical and formal approaches. By using the concept of agent autonomy as a common theme, Hexmoor gave examples of how the two approaches can complement each other in the context of various autonomy schemes. John Rushby, from Stanford Research Institute, was the next speaker. Rushby began by stressing the importance of formal methods as a means of system engineering. A mathematical model enables people to provide behavioral assurances about their system; such assurances are essential for many applications. Rushby then stated that if we design an agent as a formal method (i.e., deduction on a model) then the agent may not require external verification. Rob Axtell, from Brookings Institute, presented his view next. Axtell cautioned us to be careful in our use of formal approaches. He cited examples of potential pitfalls. The last panel member was Nenad Ivezic, from the National Institute for Standards and Technology. Ivezic, like Hexmoor, saw a complementary role for the two approaches. According to Ivezic's view, empirical approaches are good for exploration, whereas formal approaches are good for optimization and fault detection. Furthermore, the empirical community has much to learn from the formal approaches community and vice versa.

A lively discussion followed the panel members' presentations. The first question was whether formal models can incorporate imperfect knowledge. Rushby responded by saying that they can in theory but it is not clear how to do this in practice. He suggested that perhaps one solution is to delineate controlled versus uncontrolled space so that we could determine the region of reliable system performance. There was then some discussion about the need for standards, but Rushby pointed out that even standards will not solve the problems of predicting emergent behavior of complex systems. The next question was about where NASA stands on the issue of empirical versus formal approaches. Susan Torney, from Johnson Space Center, replied. She said that the answer depends on what aspect of NASA is pertinent. The greatest need for formal methods is probably in mission control of manned programs. Torney's response was followed by a lengthy discussion regarding the source and solution to NASA's recent failures. Which of these problems could have been avoided by stricter use of formal methods? Some people felt formal methods would have helped; others felt testing should also have been used.

The discussion then turned to a different topic, prompted by a question from Sheila McIlraith (Stanford University) about whether formal methods can make any assurances about probabilistic agent systems (e.g., that a system is probably approximately correct). Kristina Lerman (USC Information Sciences Institute) suggested that a solution is to use quantitative mathematics, which is somewhere between what is traditionally considered to be formal methods and more empirical approaches like testing. One could also include in the model the probability of making a wrong measurement. This discussion prompted two of the panel members to make additional comments about empirical versus formal approaches. For one, Smith suggested using formal analysis to prune large parts of the space for testing. Also, Axtell emphasized that unforeseen, unmodeled events can be the most significant problem for the applicability of formal methods. Ramesh Bharadwaj (Naval Research Laboratory) then suggested that what we really need is "empirical formal methods." He stated that the biggest problem he has found with using formal methods is that when you apply them to a real-world, complex system they have scalability problems. What practitioners need is documentation of experiences – perhaps a body of knowledge that characterizes the class of systems to which each formal method is applicable. Walt Truszkowski (NASA Goddard) stated that at NASA the question is often raised "What is the return on the investment?" and an answer to this requires empirical supporting data.

2 Empirical versus Formal Approaches to Agent-Based Systems

Henry Hexmoor
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202

Synergies exist between empirical and formal approaches to agent-based systems. We discuss these approaches and posit that whether to follow an empirical or formal approach depends on the domain of interest and the properties of the results we seek.

The focus in the empirical approach to agent-based systems is system-centric. In the empirical approach we remain grounded in objects and events of the real world. Uncertainties and other realities in the world that are difficult to formalize are best captured by empirical approaches. In domains in which observational data is available, such as biological systems, empirical approximations of real phenomena are useful. Systems of mobile robots as agents are all empirical systems and by observation we extrapolate principles, methodologies, and architectures. With the empirical approach we seek to measure, experiment, and gain access to metrics and benchmarks.

The focus in the formal approach to agent-based systems is model-centric. Cognitive ingredients of complex mental states as well as precise models and requirements of critical systems are best captured by formal approaches. Models are often designed for abstract ways of thinking about empirical objects and events. In such abstract thinking that simplifies the real world, theories are possible. Theories in turn may offer useful theorems. In environments that must provide guarantees of success, the best method is to develop formal methods that lead to theories. In such a method an inference system might be needed that is sound, complete, or rational. Using a logical formalism is a formal approach that provides properties that offer concise and expressive languages and representations.

It is conceivable that in a given problem, both approaches are useful and there is useful synergy. Let's consider an example that is guided by a formal system. Cohen and Levesque's Joint Intentions Theory [1] offers the concepts and conditions needed to form a team that works on a common goal. Following the concepts and conditions of joint intentions theory, a system can be implemented that enables team work. For example, Milind Tambe's STEAM obeys the Joint Intentions Theory and models interactions in domains like the game of soccer [2].

References

1. Cohen, P. R. & Levesque, H. J. (1990). *Persistence, Intention, and Commitment.* Cambridge, MA: MIT Press, Cambridge, MA.
2. Tambe, M. (1997). Towards flexible teamwork. *Journal of Artificial Intelligence Research 7*, 83–124.

3 Empirical Analysis of Multi-Agent Systems Models in the Social Sciences

Rob Axtell
Center on Social and Economic Dynamics
The Brookings Institute
Washington, D.C. 20036

Building a model in the mathematical social sciences typically involves specifying individual agent behavior in some domain and then "solving" for an equilibrium (e.g., Nash equilibrium) configuration. In practice it is almost always necessary to invoke one or more–usually more–simplifying assumptions in order to solve one's model. Such assumptions are typically quite unrealistic vis-a-vis humans, for example, that individuals have homogeneous behavior, or they are perfectly rational, or they can costlessly acquire and process information, or they can arrive at equilibrium, or that they interact equally with one another. Such simplifications render formal analysis tractable.

Multi-agent systems (MAS) are utilized by social scientists in order to build more realistic models of human social phenomena–financial markets, business firms, cities, social norms–by relaxing some of these "heroic" assumptions. These systems may consist of a relatively large number of agents (e.g., millions), and may have either quite simple or very sophisticated agent behavior. Typically in such systems there are either multiple equilibria, metastable states, or long-lived transient configurations of the agent population. The problem of trying to deduce the set of feasible outcomes of such models is a formidable one, for which there are no powerful techniques that work in general. Indeed, if such analysis techniques were readily available then the social sciences would be much easier, for laboratory experiments could be employed to characterize agent behavior in particular strategic circumstances, from which one could then deduce the kinds of societal behavior that are possible.

However, there are vast numbers of special cases–particular agents in specific environments–that are amenable to analysis. Some 2 $\times$ 2 games, for example, have dominant strategies for all players, games for which a Nash equilibrium may be the unique outcome, even if the exact path to such an equilibrium may be idiosyncratic. Similarly for the process of adding noise to a model, which may destabilize some equilibria and stabilize others, leading to a definite prediction of the kinds of outcomes that are possible.

Unfortunately, many ostensibly simple multi-agent models, ones that seem like their behavior should be readily deducible from agent specifications, turn out not to be. In such circumstances "empirical" methods appear to be the only route to gaining a better understanding of the overall performance of one's model. Specifically, building a MAS and making many realizations of it may be the only practical way of developing intuition about its behavior. It is a happy event when this process leads to theorems, but in many circumstances one's understanding of how a model functions will remain heuristic. Indeed, perhaps it is even the norm that, instead of resorting to empirical methods once formal analysis has "hit a brick wall," so to speak, empirical techniques are the first line of attack in analyzing the majority of MAS models built today. Almost certainly this is the case in the social sciences.

Consider an illustrative example. A (finite) population of agents is paired randomly to play the Nash demand game in three strategies. During its turn to play, an agent can ask for either most of what is to be divided (the "high" strategy), half (the "medium" strategy), or just a small amount ("low"). If the sum of the agents' demands is equal to or less than the total, their demands are met and they receive the shares they requested. If their demands are in excess of the total available they receive nothing. Further, let each agent have a (finite) memory for recent plays of its opponents, so that it can form an expectation of how future opponents will play the game, a version of "fictitious play." Finally, let each agent select the strategy that maximizes its payoff given its memory of past play, although with a small probability a strategy is selected at random.

Mathematical analysis of this game utilizes techniques from stochastic dynamical systems to prove results about its asymptotic state [2]. The main results are as follows. The state space, Z, of this game is large but finite, amounting to all possible configurations of agent memories. Agent interaction determines a Markov chain with stationary transition probabilities. As long as agents occasionally play randomly the process is ergodic and irreducible, and so has a unique stationary distribution, $m(Z)$. Asymptotically, this distribution represents the frequency the population is in state Z. Furthermore, under reasonable technical conditions, the probability the system is in the configuration in which all agents play the "medium" strategy is arbitrarily close to 1 for small but positive probability of playing randomly. That is, no matter how the system starts, if one waits long enough it will eventually end up with all agents asking for $1/2$ of the pie, this despite the fact that other Nash equilibria exist in the game as well.

At first blush these results seem both very powerful and rather surprising. Why should the agent population end up in the equity norm, in which all share equally? Why can't the society get stuck for a long time in some less equitable outcome? It turns out that empirical analysis of this model reveals that the agents can and do get stuck for long periods away from the asymptotic state, and that the transit times from inequitable to equitable outcomes increase exponentially in all the model parameters (i.e., the number of agents, the average size of agent memories, and 1/probability of playing randomly). Although the model is formally ergodic, it displays what physicists call "broken ergodicity," meaning that it can be trapped in particular configurations for times that are long with respect to the nominal period of observation or lifetime of the system.

Empirical analysis [1] has thoroughly characterized the transient behavior of this model. Together with the asymptotic, mathematical analysis we now have an essentially complete picture of the overall behavior of this MAS. In conclusion, the only systematic way to analyze realistic MAS is through a combination of formal, mathematical techniques and more empirically-oriented ones. As with most dichotomies in the real world, "formal analysis vs. empirical analysis" is a false one.

References

1. Axtell, R., Epstein, J., & Young, H. (2001). Emergence of Classes in an Multi-Agent Bargaining Model. In S. Durlauf and H. Young (Eds.), *Social Dynamics*. MIT Press: Cambridge, Mass.
2. Young, H. (1993). An Evolutionary Model of Bargaining. *Journal of Economic Theory 59(1)*, 145–168.

4 An Architecture and COTS Analysis of Agent Systems

Nenad Ivezic
Oak Ridge National Laboratory
PO Box 2008
Oak Ridge, TN 37831
and
National Institute of Standards and Technology
100 Bureau Drive
Gaithersburg, MD, 20899

4.1 Introduction

In the absence of solid software engineering knowledge, many agent-based systems development efforts are likely to fail. Clearly, principled analysis and evaluation methodologies are needed to assess agent-based systems behaviors. In this extended abstract, we outline a methodology for the early agent systems architecture analysis and evaluation with the focus on risk identification, evaluation, and mitigation. Architectural and COTS decisions on software qualities such as performance, modifiability, and security can be assessed. We describe a test-bed to analyze and evaluate agent-based systems in manufacturing enterprises based on this methodology.

4.2 ATAM and COTS Analyses for Software Systems

In large software systems, software qualities such as performance, security, and modifiability are dependent not only upon code-level practices (e.g., language choice, detailed design, algorithms), but also upon the overall software architecture. A variety of qualitative and quantitative techniques are used for analyzing specific quality attributes [1]. These techniques have evolved in separate communities, each with its own vernacular and point of view, and typically have been performed in isolation. However, the attribute-specific analyses are interdependent; for example, performance affects modifiability, availability affects safety, security affects performance, and everything affects cost.

The Architecture Tradeoff Analysis Method (ATAM) is an evolving architecture evaluation technique [4] [5]. The input to the ATAM consists of system architecture and the perspectives of stakeholders involved with that system. The ATAM relies on generating scenarios to assess the architecture, where the scenarios are stimuli used to represent stakeholders' interests and to understand quality attribute requirements. The scenarios give specific instances of anticipated use, performance requirements or growth, various types of failures, various possible threats, various likely modifications, etc.

ATAM, by itself, does not provide an absolute measure of architecture quality; rather it serves to identify scenarios of interest to diverse stakeholders (e.g., the architect, developers) and to identify risks (e.g., inadequate performance, successful denial-of-service attack) and possible mitigation strategies. The results of an ATAM evaluation reflect the concerns of stakeholders and the scenarios they consider important.

The development time and technical expertise required to create an agent framework provide strong incentive for system architects to utilize commercial off-the-shelf (COTS) agent frameworks. However, integrating COTS agent middleware into system architecture introduces risk and tradeoffs due to the vendor implementation. A key tool for integrating COTS components and mitigating this risk is the test-bed. A test-bed provides an environment for gaining product knowledge and allows system architects to gain insight into the COTS "black box" with simple tests and data collection. ATAM and COTS analyses provide complementary tool sets for designing and evaluating agent based system architectures. These tools can be combined in a variety of ways to support specific systems and system architect needs. One example of the ATAM-COTS combination is a test-bed process that defines a model problem. The model problem can be general or specific to a certain application domain. Next, an agent framework, commonly an COTS product, is chosen for evaluation in the test-bed. In addition to the agent framework, the abstract model solution is chosen. The abstract solution is an architectural pattern (or patterns) to be evaluated in the test-bed.

4.3 Agent Systems Analysis: Attributes, Architectures and Scenarios

A number of publications discussing agent-based systems contain design patterns for collaborating agents [2] [3]. However, critical aspects of evaluating agent-based systems in terms of quality attributes are not presented. We see the following attributes as particularly relevant to agent architecture evaluation:

1. Performance predictability,
2. Security against data corruption and spoofing,
3. Adaptability to changes in the environment,
4. Availability and fault tolerance,
5. Agent development productivity.

Typically, agent-system taxonomies focus on functional properties of the agents. As functionality increases, proliferation of agent "types" mushroom; confusion is inevitable because of unavoidable overlaps and subtle distinctions.

Instead of attempting to conduct attribute tradeoffs based on ever-changing taxonomies of styles, we take a different approach [5]. First, we do not look at the functionality of the agents in the system, rather we concentrate on external properties such as communication, cooperation, and coordination performed by the agent components and connections. This defines our "architecture" domain. Second, we look at the instantaneous architecture (i.e., components, connections, and the patterns of component behavior). Third, we identify a "central" component and conduct the analysis with this component as the focus of the interaction. This approach seems to describe a large variety of agent systems [5]. Example agent patterns such as wrapper, matchmaker, broker, mediator, contract-net have emerged in these analyses. Next, we devise operational scenarios for various agent types to identify and resolve potential risks.

For example, agent systems are susceptible to attacks in which an outside party intervenes in an inter-agent exchange. Attacks that require explicit knowledge of the server by the client or knowledge of the client by the server are prevented in the broker agent. On the other hand, the existence of a specific intervening agent in the communication between the client and the server might have an impact on performance (increase in latency), availability (single point of failure), and even security of information since there are now two separate communication channels to be protected. While this is unavoidable in some cases, for agent communications it is quite possible that public key encryption can help in the positive agent identification prior to acceptance of a particular message. Such choices are the nature of architectural analysis.

4.4 A Test-Bed for Agents System Analysis

A test-bed concept is under development at the National Institute of Standards and Technologies (NIST) to address the need for repeatable, disciplined, and readily available techniques to measure the effectiveness of different agent technologies. The test-bed will include metrics ranging from simple facts (e.g., literature, number of users) to sophisticated analyses (e.g., ATAM, COTS). The test-bed will collect the outcomes of applying particular agent technologies to various scenarios. We expect that many scenarios could be provided from the agent research and the user communities. Example problems of varying detail could provide a common communication medium between researchers, vendors, and users of agent technologies. Scenario application should include a variety of qualitative and quantitative techniques, ranging from discussions that follow from a scenario exploration, to a model problem and a discussion on how that model might be analyzed, all the way to a formula to calculate the value of a particular quality attribute. The test-bed initially focuses on business-to-business, supply chain management application domain within which emerging standards (e.g., RosettaNet) will be used to drive the architectural analyses of agent systems.

References

1. Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). *Quality Attributes* (CMU/SEI-95-TR-21, ADA307888). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
2. Brenner, W., Zarnekow, R., & Wittig, H. (1998). *Intelligent Software Agents, Foundations and Applications.* Berlin, Germany: Springer-Verlag.
3. Hayden, S. C., Carrick, C., & Yang, Q. (1999). Architectural Design Patterns for Multi-Agent Coordination. In *Proceedings of the International Conference on Agent Systems. (Agents'99).* Seattle, WA.

4. Klein, M. & Kasman, R. (1999). *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
5. Woods, S. G. & Barbacci, M. R. (1999) *Architectural Evaluation of Collaborative Agent-Based Systems* (CMU/SEI-99-TR-025) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Author Index

Panel Discussion: "Future Directions"

1 Moderator's Summary of the Panel Discussion

Michael Hinchey
NASA Goddard Space Flight Center (Code 588)
Greenbelt, MD 20771

The FAABS'00 panel on "future directions" generated some interesting discussion and even some heated debate. The aim was to consider in what directions the development of agent-based systems might lean in the future, and in particular the role of formality in this.

James Hendler begins by describing DARPA's TASK initiative, which is exploring applying not just formal approaches, but a diverse range of mathematics, to modeling agent-based systems.

Charles Pecheur emphasizes the need for formal approaches in the early stages of development. That is, formal design, which can bring many benefits.

Connie Heitmeyer considers the classes of formal languages and formal analysis techniques and tools that may be appropriate for agent-based systems, highlighting a number of research issues that these give rise to.

Diana Gordon sees two deficiences in current formal approaches. The first is that little attention is paid to scalability and synthesis of very large multi-agent systems. The second is lack of consideration of adaptability, predictability, and timeliness in the context of agents that adapt by learning.

Michael Luck sees the need for greater links between formal theories and empirical research. He too highlights the need for appropriate languages, and in particular languages appropriate for multi-agent systems.

Walt Truszkowski points to the need for formalisms that ensure consistency in multi-agent systems that learn. He points to the likelihood of needing several formal models in order to address different aspects of more complex agent-based systems, and asks whether formal methods in the future will provide sufficient scope to consider systems in which human beings are an integral component.

While the various panelists have varying viewpoints, there is an underlying consensus: Future research must address the further development of appropriate formal languages and approaches that can scale to large multi-agent systems that adapt by learning. It is far from certain that existing approaches alone will be successful in this goal.

2 Formalizing Agent-Based Computing: DARPA's TASK Program

James Hendler
Program Manager
Defense Advanced Research Projects Agency
and
Department of Computer Science
University of Maryland
College Park, MD 20742

In the complex realm of modern military operations, commanders are dealing with increasingly diverse missions, including operations other than war, expeditionary missions, and controlling dangerous situations in dynamic and uncertain environments. All of these missions are further complicated by the requirement for joint and coalition coordination. Achieving decision superiority in these situations is becoming increasingly difficult and complex. The need to gain the right information at the right time for each type of decision maker in this complex scenario is leading the Department of Defense (DoD) toward distributed information systems that are managed and accessed in a network-centric manner.

A key technological innovation capable of handling the complexity of modern warfare is that of software agents. Agent-based computing focuses on the development of distributed computational entities (software agents) which can act on behalf of, mediate or support the actions of human users and autonomously carry out tasks to achieve goals or assist the activities of the users in achieving those goals. In the military, using these agents will improve our information and decision management capabilities and thus drastically reduce the complexities of modern warfare.

Unfortunately, developing agent-based software is currently something of a "black art" – an arbitrarily complex process that has many ad hoc methodologies but no significant modeling solutions. Very few rules or protocols have been agreed on for analyzing and modeling complex, large-scale, agent systems that interact with distributed heterogeneous information systems. As a result, DARPA is investing in understanding the principles underlying agent-based computing via its Taskable Agent Software Kit (TASK) initiative.

The TASK program's goal is to extend the current scientific and mathematical foundations of agent-based computing and to add precision to the engineering of agent-based systems. TASK researchers will explore the application of mathematics not traditionally used for computer modeling, such as statistical physics and chaotic dynamics, to explore the modeling of multi-agent systems.

TASK researchers will demonstrate their algorithms and approaches against a set of "research exploration frameworks" chosen to be of great relevance to a very wide range of DoD problems. Specific focus will be placed on dynamic command and control problems, the fundamentals of cooperative systems, and understanding information-agent behavior. In this way, new techniques for resource assignment against chaotic problems, hybrid control of dynamic systems, and modeling the information dynamics of complex IT systems will be explored and exploited.

For more information about the TASK program and its goals, participants, and approach see our website at http://www.task-program.org.

3 Future Directions

Charles Pecheur
RIACS / NASA Ames Research Center
Moffett Field, CA 94035

My arguments are articulated around the four questions provided as guidelines for this panel. My perspective is that of someone doing applied research in formal verification for NASA, though I tried to keep an open mindset in this exercise.

What aspects of agents should be formalized? In the formal methods community, the focus is on formal *verification*, for the sake of improving and asserting the *reliability* of a design. However, formal methods, in their broader meaning of mathematically sound software engineering techniques, have a big role to play at the *design* stage as well. After all, every software produced now and in the past is already based on some formality, in the form of programming languages and compilers. A main issue is to make future design methods not only *more formal* but also *more abstract*. In term, this should simplify the task of the programmer and therefore increase his productivity. Furthermore, a more abstract formal design is also more easily amenable to formal verification.

Although this trend is already widely endorsed in the research community, formal methods are still perceived as adding cost that can only be justified by high reliability requirements. I believe that improvements in performance, tools, practices and training will some day make formal design *cost-effective* too. Remember there were times when no one would think of writing an operating system in anything else than assembly language...

What applications would show the value of formal approaches to agent-based systems? As a corollary of the preceding argument, formal approaches could show their benefits not only on safety-critical applications (bio-medical, transportation, avionics, space; NASA has many) but also in the rapid and cost-effective development of custom agent-based applications. A typical example would be a team of general-purpose robots that could be easily and flexibly re-configured to accomplish a new set of tasks.

What do you consider important for people to be working on, and why? Agent-based systems cumulate many features of software design: they are commonly concurrent, reactive, real-time or even hybrid, AI-based and sometimes adaptive. Formal approaches are needed to master all those aspects, but each of these features increases the complexity of formal verification by orders of magnitude.

Obviously, more CPU and RAM will always help. More specifically, formal verification of real-time and hybrid systems is a field of active ongoing research. Different prototype techniques have been demonstrated, but a lot of work is still needed to scale them up to real-size applications. Verification of adaptive systems is even more problematic, considering that such systems can modify their own behaviour. Verification could be addressed after or before the adaptation has occurred, and different kinds of adaptation (e.g. neural nets vs. knowledge-based systems) will surely ask for different verification techniques.

Even well-mastered verification technologies such as finite-state model checking, though, still need to be integrated into a software development methodology. There are now well-established practices for managing a million-line software project, controlling its quality, measuring its progress, predicting its cost. Formal methods have to be inserted into that framework, with experimental evidence of their usefulness at the global level of a product life-cycle.

What new ideas on formal approaches would be applicable to agents? Model checking is based on exhaustive exploration of the states of a system. This is conceptually appealing but practically intractable except for simplified models, despite progress in computer performances and model checker sophistication. Because of this, the bulk of the work is actually to do that simplification, and only verification experts, not software developers, can afford to do it. This is the main reason for the poor acceptance of formal verification methods by the software industry.

Instead of thriving to force pieces of big programs into existing model checkers, an alternative is to add pieces of model checking into big program development tools that developers use routinely. For example, a source-level debugger could be enhanced to support backtracking and alternative exploration of the different executions of a concurrent program. Program states could be stored and compared to detect loops. This would not provide any guarantee of correctness, but it would be a useful and significant extension of the debugging capabilities. And if it sits as a new button in their usual toolset, developers will be much more likely to start to use it. Tools like VeriSoft (Lucent) and Java Path Finder 2 (NASA Ames) pertain to that philosophy. They allow automated exploration of programs written in C and Java, respectively.

4 Research Issues in Agent-Based Systems: The Role of Formal Methods

Constance Heitmeyer
High Assurance Computer Systems Branch
Naval Research Laboratory (Code 5546)
Washington, DC 20375

4.1 Introduction

Jennings, Sycara, and Wooldridge [7] describe an agent as a "computer system, situated in some environment, that is capable of flexible, autonomous action in order to meet its design objectives." Agents are being used increasingly in a wide variety of systems, ranging from autonomous vehicles to robots to e-mail filters. A number of specialized logics and formal models have been developed to describe and to reason about agents. Hence, it makes sense to ask what research is needed in customizing and applying more standard *formal methods*, such as formal languages and formal analysis techniques, to agents.

4.2 Languages for Describing Agents and Their Properties

One set of research issues concerns the formal languages that are used to describe agents and the protocols that the agents use to communicate. Specific issues include the following:

- Should generic agent command languages, such as KIF (Knowledge Interchange Format) and KQML (Knowledge Query and Manipulation Language) [3], be used to describe agents?
- Alternatively, should domain-specific languages be developed?

Another set of issues concerns the classes of properties one wants to prove about agents. For example, agent behavior may be checked for completeness (a behavior is specified for every possible situation) and consistency (each specified behavior is unambiguous). Other properties that may be checked include safety, fault-tolerance, and freedom from deadlock.

4.3 Formal Techniques for Analyzing Agents

Three major classes of formal techniques have emerged for analyzing systems and system designs for properties of interest. The first class contains the set of mechanical theorem provers, such as PVS [9] and HOL [4], which allow a user to use deductive reasoning to prove formally that a design satisfies some property of interest. To make such provers easier to use, specialized interfaces, such as TAME (Timed Automata Modeling Environment) [1], have been developed that facilitate reasoning about classes of systems using specialized theories and strategies.

A second class of analysis techniques contains model checkers, such as Spin [6] and SMV [8]. In model checking, the user represents the system as a finite state machine and then invokes the model checker to check that the model satisfies a selected property. Unlike theorem provers which are designed to *verify* system properties, model checkers have mostly been used to detect property violations. Compared to traditional theorem provers, whose use requires theorem proving skills and detailed knowledge of the theorem prover (e.g., its language and base logic), model checkers are relatively easy to use. However, due to the large state space of models of practical systems, users often need ingenuity to reduce the state space of the model. Without such reductions, model checking is usually infeasible.

A third class of formal analysis techniques which have emerged recently are based on a complete decision procedure. Because their underlying logic is decidable, such techniques perform the formal analysis completely automatically. Examples of tools in this class include Salsa [2], which decides the validity of formulae composed of logical propositions and linear integer constraints, and Mona [5], which handles monadic second-order logic of one successor.

A number of research issues concern the role of formal analysis tools in developing agents. These issues include the following:

- Given a class of agent properties, which analysis tool is most suitable for reasoning about the property?
- How should formal analysis tools be customized for reasoning about agents?
- What new tools are needed to reason formally about agents?
- How should existing tools be combined with each other and with other tools, e.g., simulators, to reason about agents?

References

1. Archer, M., Heitmeyer, C., & Sims, S. (1998). TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, Netherlands. Eindhoven University Technical Report.
2. Bharadwaj, R. & Sims, S. (2000). Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking. In Graf, S., Schwartzbach (eds.): *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Sciences, Vol. 1785*. Berlin: Springer-Verlag.
3. Genesereth, M. R. & Ketchpel, S. P. (1994). Software Agents. *Comm. of the ACM 37(7)*.
4. Gordon, M. J. C. & Melham, T. (1993). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press.
5. Henriksen, J. G. et al. (1995). Mona: Monadic Second-Order Logic in Practice. In *Workshop on Tools and Algorithms for the Construction of Systems*, Aarhus.
6. Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*, Prentice-Hall.

7. Jennings, N. J., Sycara, K., & Wooldridge, M. (1998). A Roadmap of Agent Research and Development, *Autonomous Agents and Multi-Agent Systems 1.*
8. McMillan, K. L. (1993). *Symbolic Model Checking*, Kluwer Academic Publishers.
9. Shankar, N., Owre, S., & Rushby, J. (1993). *The PVS Proof Checker: A Reference Manual*, Computer Science Lab., SRI Intl., Menlo Park, CA.

5 Future Directions for Agent-Based Systems

Diana Gordon
AI Center, Naval Research Laboratory (Code 5514)
Washington D.C. 20375

5.1 Introduction

Although the topic of this workshop, "Formal Approaches to Agent-Based Systems," is relatively new, there were many submissions, which reflects the significance of the topic. The driving force behind this blossoming new field is a set of applications that demand the flexibility of an agent-based approach coupled with the behavioral assurance provided by formalism.

Furthermore, the eclectic nature of the approaches presented at the workshop suggests that there are a variety of ways to address this important topic. Although the majority of participants had a background in formal methods from the field of computer science, some participants came from other mathematically-oriented fields, such as economics and physics. Therefore, we use the term "formal" in its broadest sense here – to denote any mathematical technique.

Most of the workshop papers focused on a few important questions, e.g., How does one methodically synthesize agents? What is the best way to transition agents from design to implementation without losing rigor? What methods enable successful multi-agent coordination? Can we improve inter-agent communication in a principled manner? Which applications have the characteristics that are well-suited to the field of formal approaches to agent-based systems, and how can we address problems that arise in the context of these applications? Certainly these questions are important to address, and future work should continue to address them. On the other hand, two very significant open problems were under-represented at the workshop. The remainder of this discussion will focus on these two issues.

5.2 Achieving Desirable Aggregate Behavior

Many applications of the future will require coordination of large collections of agents. Examples include teams of software agents performing data mining or facilitating electronic commerce, large clusters of tiny space vehicles forming a remote virtual telescope, or large numbers of micro-air vehicles assembling themselves into a virtual antenna. In all of these cases there is a common set of problems to be addressed. For example, what is the best configuration of the agents to solve the task? What control mechanisms can achieve desired global configurations while maintaining stability, fault-tolerance, and/or self-repair?

The degree of complexity in large multi-agent systems can be overwhelming. To glean insights into how to achieve desirable aggregate behavior will most likely require a clever combination of empirical data analysis and mathematical modeling. Some papers (e.g., [9]) at the workshop addressed the issue of multi-agent coordination via model checking of global multi-agent properties, and one paper [7] used a game theoretic approach. Nevertheless, the workshop lacked papers about how to synthesize desirable aggregate behavior in *very large* multi-agent systems, e.g., with hundreds of agents. Approaches elsewhere in the literature include the imposition of social laws on societies of agents [10] or artificial physics forces for distributed agent control [11]. Two papers at the FAABS'00 workshop did in fact address a related topic. They presented mathematical techniques for *analyzing* very large multi-agent systems. The first, by Axtell [1], introduced a novel method for analyzing emergent global behavior based on analyses of social structures, such as firms or markets. The second, by Lerman [4], uses differential equations to model the macroscopic phenomenon of coalition formation based on microscopic interactions.

Given the importance of constructing desirable aggregate behavior and analyzing it, and given that the focus of these workshops is on bringing formal approaches to bear on problems, these topics should be better represented in subsequent FAABSs workshops. One of the most significant open challenges is to investigate which formal methods are applicable. The following are just a few examples of techniques that have already proven successful at modeling natural systems composed of a huge number of individual elements: Differential equations for epidemiological modeling of virus spread, statistical mechanics for analyzing thermodynamic properties of particles, and biology for understanding complex multi-cellular organisms. They also appear to be applicable to agent-based systems, but further research is needed to compare their advantages and limitations in the context of agents. It would also be useful to explore the applicability of theorem proving and other logic-based formal methods to the study of emergent collective behaviors in large systems.

It is hoped that future work will focus on the application and comparison of existing mathematical methods, coupled with the development of novel techniques, to address the synthesis and analysis of large multi-agent systems with desirable aggregate behavior.

5.3 Agents That Are Adaptive, Predictable and Timely

The previous topic, aggregate multi-agent behavior, assumed non-learning agents. What about agents that adapt by learning? We need our agents to be adaptive so that they can handle unforeseen conditions. We also need them to be predictable so that we trust their behavior, as well as timely so that their responses are not outdated. How can we simultaneously satisfy all three of these requirements? Although these requirements appear to be contradictory (e.g., adaptation could reduce predictability; verification and repair to increase predictability could be time-consuming), they are all important. Three of the papers in this workshop [2] [3] [6] presented ideas on how one might address all of these requirements in one (multi-)agent paradigm. These papers presented two solutions: learning (adaptation) methods that are property-preserving and thus do not require reverification, and efficient algorithms for reverification after learning. Nevertheless, this work only covers a small subset of the set of possible approaches. In particular, the work assumes specific choices for the agent architectures, property classes, learning algorithms, and formal methods.

A potentially fruitful future direction would be to explore how to achieve adaptivity, predictability, and timeliness in the context of a variety of different agent architectures or classes of relevant agent properties/constraints. For example, Gordon [2] assumes agent strategies are represented as deterministic finite-state automata. However there are many other options, such as stochastic finite-state automata, rule sets, or neural networks. Quite a variety of possibilities were presented at the FAABS'00 workshop, including the popular belief-desire-intention (BDI) framework for agent design. For other ideas on possible architectures, one could consult the proceedings of the workshops on Agent Theories, Architectures, and Languages (ATAL). Regarding classes of properties, Manna and Pneuli [5] provide an excellent taxonomy of temporal logic properties. Stochastic and timed properties are also possibilities.

One of the most profitable directions for future research would be to explore the wide variety of machine learning methods used by agents [8]. The list of possible learning techniques is broad and includes statistical learning, case-based inference, reinforcement learning, Bayesian updating, evolutionary learning, inductive inference, speedup learning, and theory refinement. These learning methods enable adaptation to changing environmental conditions, and they can also increase the efficiency and/or effectiveness of strategies used by agents. For further ideas, see the Learning in Multi-Agent Systems Webliography at the URL http://dis.cs.umass.edu/research/agents-learn.html.

Finally, Gordon [2], Kiriakidis and Gordon [3], and Owre et al. [6] assume that the formal methods used are model checking, discrete supervised control, and theorem proving, respectively. Other formal methods should be considered as well. In fact, in general the field of formal approaches to agent-based systems should remain eclectic – so that we can be problem-oriented and thereby seek the most suitable method for solving our agent problems.

References

1. Axtell, R. (2000). A formal theory of emergence. *FAABS'00 Abstracts.*
2. Gordon, D. (2001). APT Agents: Agents that are adaptive, predictable and timely. *FAABS'00 Proceedings.*
3. Kiriakidis, K. & Gordon, D. (2001). Adaptive supervisory control of multi-agent systems. *FAABS'00 Proceedings.*
4. Lerman, K. (2001). Mathematical modeling of coalition formation in large multi-agent systems. *FAABS'00 Proceedings.*

5. Manna, Z. & Pnueli, A. (1989) The anchored version of the temporal framework. *Lecture Notes in Computer Science 354.* Berlin: Springer-Verlag.
6. Owre, S., Rueβ, H., Rushby, J., & Shankar, N. (2000). Formal approaches to agent-based systems with SAL. *FAABS'00 Abstracts.*
7. Rudnianski, M. & Bestougeff, H. (2001). Modeling task and teams through game theoretic agents. *FAABS'00 Proceedings.*
8. *Proceedings of the Workshop on Multiagent Learning (1997).* S. Sen (ed.) AAAI Workshop Proceedings.
9. Shapiro, S., Lesperance, Y., & Levesque, H. (2001). Modeling and verification with the cognitive agents specification language. *FAABS'00 Proceedings.*
10. Shoham, Y. & Tennenholtz, M. (1992). Social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73.
11. Spears, W. & Gordon, D. (1999). Using artificial physics to control agents. *ICIIS'99 Proceedings.*

6 On Agent-Specific Formal Methods

Michael Luck
Department of Computer Science
University of Warwick, Coventry CV4 7AL, UK

6.1 Introduction

Over the past ten years, the field of agent-based systems has grown rapidly and dramatically to become one of the most dynamic and significant areas of computer science. Perhaps as a result of this speed of progress, the field has developed along two largely distinct lines, corresponding to empirical and formal research. While there is beginning to be some interaction between these two areas, there is still a danger of fragmentation into isolated groupings. Both approaches are, however, vital to sustained progress in work in agent-based systems. Arguably, the most important aspect for the future of formal approaches to agent-based systems lies in its ability to inform and contribute to practical systems development. In that context, this contribution reviews the role of formal approaches to agent-based systems to date and outlines some further directions, describes some concerns that are likely to provide further impetus to future work in the area, and highlights the overarching principles that must direct such work.

We can begin by summarising the aim of work in agent-based systems as being to build systems capable of flexible and autonomous decision-making, with multiple systems interacting and cooperating with each other. Formal work in support of this goal of agent-based systems has reflected more general work in artificial intelligence by being focussed in three key ways [1].

- First, logic has been used to axiomatise common-sense reasoning such as Cohen and Levesques's theory of intention, and the more practically focussed BDI theories of Rao and Georgeff [5].

- Second, logic has been used as a knowledge representation language, for example to provide formalisms for modelling belief (eg [3]).
- Finally, logic has been used as a programming language to implement agents, with examples including Fisher's work on Concurrent MetateM [2].

6.2 Formality in Relation to Agents

In the context of agents, we can consider the contribution that formal methods can make in support of the further development of the field. First, we can describe the uses to which we might put formal approaches to investigate and underpin the foundations of the field and, second, we can describe the agent applications that demand the use of formal approaches and provide the impetus for work at the intersection of both fields. Each is considered briefly below.

Agent Requirements In relation to agent-based systems in particular, there is a need for formal theories to express such components as perception, action, knowledge, belief, goals, motivation, desire, intention, and so on. The difficulty with representing such components is that while it provides useful agent theories, it is likely to produce complex theories that are not tractable. We also need languages with appropriate abstractions for programming agents — high-level languages such as Java are inadequate for a specifically agent approach because they don't support the particular kinds of abstractions that give value to the agent metaphor. These new languages can then also provide a way of implementing formal agent theories. Finally, theories and languages for *multi-agent* systems are needed in addition to single-agent systems, incorporating multiple agents, common knowledge, joint intention, cooperation, coordination and communication.

Typically, efforts to address these concerns have produced formalisations of idealised or abstracted systems so that there is still a significant gap between the theory and implementations. Narrowing that gap is a key area of current work.

Agent Applications On a related note, and as pointed out by d'Inverno et al. [1], "at least 90% of the next 700 formalisms for reasoning about agents will have no impact whatsoever on the development of the field." Yet many applications may demand a formal analysis, in particular areas of concern that are additional to the more general issues relevant to mission-critical systems as a whole. For example, agents have been touted as an effective means of interaction in e-commerce and e-business systems. Here, agents might negotiate on behalf of a user or organisation, or they might participate in auctions, or otherwise engage in transactions on behalf of others. In such cases, users will need assurances about agent behaviour that can be provided by formal approaches. In a similar respect, perhaps the biggest barrier to the uptake of *mobile* agents has been concerns over security; when your mobile agents execute at remote hosts, or when other mobile agents execute at your host, security guarantees against malicious behaviour, for example, will be necessary. Despite the suspicion over formal approaches, these kinds of concerns are likely to drive forward work on the application of formal approaches to agent-based systems.

6.3 Formality in Support of Practice

More generally, formal agent theories can be regarded as agent specifications; they describe and constrain agent behaviour, and provide a base from which to design, implement and verify agent-based systems. The last of these, verification, has been the traditional focus of formal approaches, but the first two concerns of design and implementation are also vital and can benefit from the incorporation of formal methods. Indeed, there is a need for formal approaches to support the movement from specification to implementation, and for formal theories to be guided by the practical applications of agents.

In particular, for work on the development of agent-based systems to be sustained, accessible and easy-to-use methodologies for the refinement and implementation of formal agent specifications, for example, are required. This might be achieved through the adaptation of existing techniques such as UML to cope with agents, or through methods similar to the familiar routes offered by Z specification and refinement for agents. Some current work seeks to do exactly this in analysing the key characteristics of agent-based systems and then trying to extend existing methods to deal with the particular agent aspects that are not adequately addressed already (eg. [4]). In this way, the concerns of empiricists and formalists can be brought together, to support practical development and deployment in pursuit of the ultimate goal of building systems.

6.4 Conclusion

The value of formal approaches to agent-based systems lies in their contribution to the broader goals of building agent systems. This is not to say that system building is the only concern, but for the continued success of the agent paradigm, real-world practical problems must be addressed. With agents, there are numerous theoretical issues to consider, but there is also a whole new set of problems that arise when considering systems that are delegated to act on behalf of others in foreign environments. Certainly there are theoretical problems here, but the commercial imperative for such systems today demonstrates a real need for practical solutions that may be delivered by formal methods. In all of these areas, however, there are two key questions that must be considered. First, what is unique about agents and multi-agent systems that has implications for formal approaches? Second, how can we tailor formal approaches to deal with agents, be that through the development of new techniques, or the adaptation of existing ones? While the discussion above suggests some ways in which the future of the field might develop, these questions suggest many others.

References

1. d'Inverno, M., Fisher, M., Lomuscio, A., Luck, M., de Rijke, M., Ryan, M. & Wooldridge, M. (1997). Formalisms for multi-agent systems. *Knowledge Engineering Review*, 12(3): 315–321.
2. Fisher, M. (1995). Representing and executing agent-based systems. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI 890)*, pages 307–323. Springer.
3. Konolige, K. (1986). *A Deduction Model of Belief.* Pitman: London.
4. Luck, M., Griffiths, N., & d'Inverno, M. (1997). From agent theory to agent construction: A case study. In *Intelligent Agents III (LNAI 1193)*, pages 49–63. Springer.
5. Rao, A. S. & Georgeoff, M. P. (1992). An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowedge Representation and Reasoning*, pages 439–449.

7 Three Challenges for Formal Methods and Models

Walt Truszkowski
Senior Technologist
NASA Goddard Space Flight Center (Code 588)
Greenbelt, MD 20771

"Prediction is difficult especially about the future." *Niels Bohr*

7.1 Introduction

As part of the Panel Discussion on Future Directions I would like to identify a question for each of three areas of work that I feel will be important for the future of formal methods. These areas are:

– Adaptability
– Granularity
– System scoping

I will briefly discuss each in more detail.

7.2 Adaptability

Some systems can vary over time. This is especially true for agent-based autonomous systems. Intelligent agents evolve over time based on many factors including being embedded in a changing environment and having to adjust to it, and having the capability to learn based on self assessment against performance criteria established for the agent. Formally stated:

Question 1. Given that $M(S_t) = \mathrm{T}$ (i.e., the model of system S at time t is a correct model) what can formal modeling do to allow one to evolve the M

to ensure that $M(S_{t+1}) = \mathrm{T}$ where S has changed over time because of system environmental changes or “learning”?

Is there a tried and true way to evolve the model M to reflect the changes the system S has undergone because of adaptation to environmental changes or the system’s learning process?

7.3 Granularity

The next question is related to what I call granularity. As systems get “more complicated” a single formal method or model may be inappropriate or not feasible for the system as a whole. System decomposition into components and the individual modeling of each component by an appropriate formal technique tailored to that component may be required. Multiple differing formal and/or heuristic models may be required to address the various nuances of the system’s components. The question is: will an algebra of models be available to allow composition of the individual component models into a model of the entire system, i.e., will we have

$$M(S) = M_1(S_1) + M_2(S_2) + ... + M_n(S_n)?$$

7.4 System Scoping

In classical cybernetic theories, a system has to have a definite boundary that delimits it from its environment. The question, which I raise regarding system scoping, has to do with what is included within the boundary. The following two-sided figure illustrates one instance of the question regarding scope.

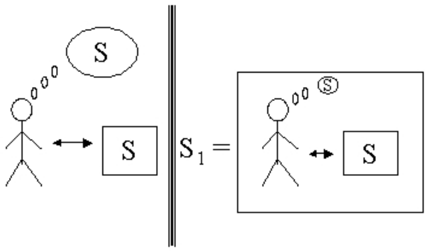


Fig. 1.

The left-hand side illustrates a person interacting with a system S according to his mental model of S. The right-hand side shows the user and his mental model as being an integral part of a larger system S_1. Formal methods today address the modeling of system S. Will formal methods be able to address the modeling of system S_1, i.e., systems in which the human is an integral component?

7.5 Conclusions

This brief panel presentation has addressed three question which I feel will need to be addressed if formal modeling techniques are to make an impact on future complex systems modeling.